



HISTORY-BASED ACCESS CONTROL FOR XML DOCUMENTS

Dissertationsschrift

in englischer Sprache

vorgelegt

am Fachbereich Informatik

der Technischen Universität Darmstadt von

Dipl.-Inform. Patrick Röder

geboren am 6.6.1978 in Singapur

zur Erlangung des akademischen Grades

Doktor-Ingenieur

– Dr.-Ing. –

Gutachter: Prof. Dr. Claudia Eckert

Prof. Dr. Rüdiger Grimm

Tag der Einreichung: 1. Oktober 2007

Tag der Disputation: 7. April 2008

Darmstadt 2008

Hochschulkennziffer D 17

Kurzfassung (Deutsch)

In den letzten Jahren erfolgte ein Wandel von Dokumenten auf Papier zu elektronischen Dokumenten, welche auf Computern verarbeitet werden. Neben Vorteilen bringen elektronische Dokumente aber auch neue Risiken mit sich. Sie lassen sich zum Beispiel einfacher stehlen und die Gefahr von absichtlicher oder versehentlicher Löschung ist größer als bei Dokumenten auf Papier. Um diese Gefahren zu reduzieren wurde eine Reihe von technischen Maßnahmen entwickelt. Zu diesen Maßnahmen gehört auch die Zugriffskontrolle. Bei der Zugriffskontrolle wird definiert, wer unter welchen Umständen auf welche Daten zugreifen darf. Durch Zugriffskontrolle kann man die zuvor beschriebenen Risiken reduzieren, indem man die erlaubten Zugriffe auf die notwendigen beschränkt.

Es gibt viele verschiedenen Möglichkeiten ein Zugriffskontrollsystem zu gestalten. Die einzelnen Möglichkeiten unterscheiden sich unter anderem in der Art der Daten, auf die der Zugriff kontrolliert wird. Diese Arbeit behandelt ein Verfahren für die Zugriffskontrolle auf XML-Dokumente, da XML ein weit verbreiteter Standard für den Austausch von Daten geworden ist und eine Reihe von technischen Vorteilen gegenüber anderen Formaten bietet.

Ein weiteres Unterscheidungskriterium ist, wie erlaubte oder verbotene Zugriffe definiert werden. Bei einfachen Verfahren muss dies für jedes Objekt manuell definiert werden. Aufwändigere Verfahren können die Zugriffsrechte aus den Eigenschaften der Objekte ableiten. Hierfür wird in der Regel ihr Dateninhalt herangezogen, aber nicht ihre Herkunft oder Entstehungsweise. Diese Informationen werden in dieser Arbeit als “History” bezeichnet und bei der Bearbeitung eines Dokumentes aufgezeichnet.

Mit Hilfe der History lässt sich der Schutzbedarf von Objekten präziser definieren, da neben dem Dateninhalt auch die Art und Weise der Erstellung herangezogen werden kann. Sind Objekte beispielsweise durch Kopieren entstanden, lässt sich der Ursprung der Kopie bei Definition der Zugriffsrechte mit berücksichtigen. Ebenso kann in Betracht gezogen werden, wer ein Objekt erstellt hat, wann es erstellt wurde und welchen Inhalt es früher einmal hatte. All diese zusätzlichen Eigenschaften erlauben es, die Zugriffsrechte

für Objekte präziser zu spezifizieren, da diese Eigenschaften eine weitere Differenzierung ermöglichen.

Für diese Art von Zugriffskontrolle entwickelt die vorliegende Arbeit ein Modell, eine Systemarchitektur sowie die benötigten Sicherheitsmechanismen.

Abstract (English)

In the recent years, there was a development in which paper documents were more and more replaced by electronic documents. As electronic documents introduce several advantages, they also bring new risks. Electronic documents can be stolen more easily and its also happens easier that they are deleted accidentally.

Access control mechanisms were introduced, to reduce this risks, which is achieved by restricting all accesses to the required ones. Many different solutions were developed to provide access control for different types of storing data. Since XML is widely used and offers many technical advantages, we focus on access control for XML documents in this thesis.

Previous methods for access control defined access for each object individually and manually, which is error-prone and time-consuming. Advanced approaches derive the access rights from the properties of an object. They use the content of an object to derive the required level of protection and finally its access rights. An essential property, which has not been regarded, is how objects have been created. We refer to this as the “History” and record it while a document is edited. This history allows us to derive the access rights more precisely, since it adds a further way of differentiating between objects. For example, it can be regarded from where an object was copied, when it was created or by whom it was created. Furthermore, is the previous content of a document can be regarded to define the access rights of a documents.

In addition to our history-based access control model, we present a system architecture for it and describe the security mechanisms for the architecture.

Acknowledgements

First of all, I greatly thank my supervisor Prof. Dr. Claudia Eckert for giving me the opportunity to do my research and for giving me all the freedom that I need to do this. I acknowledge her for always being available when I needed her advice. Moreover, she has created a relaxed atmosphere in our research group, which makes researching very comfortable.

My colleagues at our research group “IT Security” provided both a pleasant and stimulating working environment. We had a lot of fruitful discussions and my colleagues were always available when I needed any kind of advice. In particular, Omid Tafreschi helped me a lot and it was a great pleasure for me to work in several projects together with him. Moreover, I would like to thank Fredric Stumpf for a great time during our common research. I also want to thank Sascha Müller and Christoph Kraus for being very pleasant office mates.

I thank my parents Fridolin and Monika Röder and the rest of my family for their great support during all the years of my academic and non-academic life. In addition to my family, my best friend Christoph Müller and my girlfriend Heike Schmidt provided me with mental support that was very helpful for me.

Last but not least, I thank the DFG (“Deutsche Forschungsgemeinschaft”) who financed my work within the PhD program *Enabling Technologies for Electronic Commerce*. Due to their support, I was able to concentrate very focussed on my research.

Wissenschaftlicher Werdegang

6.6.1978 Geburt in Singapur

1985 - 1989 Besuch der Grundschule “Deutsche Botschaftsschule” in Kuwait

1989 - 1998 Abitur am Gymnasium “Lichtenbergschule” in Darmstadt

1999 - 2004 Informatikstudium an der Technischen Universität Darmstadt

2004 - 2008 Promotion in Informatik an der Technischen Universität Darmstadt

Declaration

These doctoral studies were conducted under the supervision of Prof. Dr. Claudia Eckert. The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled in the Department of Computer Science as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Darmstadt, April 12, 2008

Contents

Kurzfassung (Deutsch)	iii
Abstract (English)	v
List of Figures	xvii
List of Tables	xix
List of Algorithms	xx
1 Introduction	1
1.1 Motivation	1
1.2 Structure of this Thesis	8
2 Background	11
2.1 Access Control	11
2.1.1 Access Matrix Model	14
2.1.2 Role-Based Access Control (RBAC)	15
2.1.3 Chinese Wall model	18
2.2 The Extensible Markup Language	21
2.2.1 Introduction to XML	21
2.2.2 XML Documents	22
2.2.3 XML Namespaces	23
2.2.4 XML Schema	24
2.2.5 XML Path Language (XPath)	25
2.3 Trusted Computing Group Mechanisms	26
2.3.1 Remote Attestation	28
2.3.2 Integrity Reporting Protocols	28
3 Scenario and Requirements	31
3.1 Scenario	31
3.2 Requirements	36

4	Model	41
4.1	Overview	41
4.2	Subjects	42
4.3	Objects	42
4.3.1	XML elements	42
4.3.2	Attributes	43
4.3.3	Text blocks	43
4.4	Operations	44
4.4.1	Create	44
4.4.2	Delete	45
4.4.3	Copy	46
4.4.4	Change Attribute	47
4.4.5	View	47
4.5	History	48
4.6	Access Control Rules	50
4.6.1	Role field	50
4.6.2	Operation field	51
4.6.3	Object field	51
4.6.4	Destination field	51
4.6.5	Mode field	52
4.6.6	Conflict resolution strategy	52
4.6.7	Default semantics	52
4.7	Accessing the History with XPath	53
4.7.1	Getting Copies of an Object	53
4.7.2	Getting Related Nodes Depending on Time	55
4.7.3	Getting the Context of a History Entry	56
4.7.4	Getting Accessed Nodes	57
4.7.5	Getting Specific Nodes of Current Rule	57
4.7.6	Additional Extension Functions	58
4.8	Modeling Chinese Wall policies	58
4.9	Summary	62
5	System Architecture	65
5.1	Architecture Overview	66
5.2	Workflow	67
5.2.1	Check-out	68
5.2.2	Editing	68
5.2.3	Check-in	74
5.3	Distributed System Architecture	75
5.3.1	Overall Approach	76
5.3.2	Client-Server Approach	78

5.3.3	The scalability of the distributed system architecture . . .	85
6	Security Architecture	87
6.1	Risk Analysis	87
6.1.1	Attacks on the Client Machine	89
6.1.2	Attacks on the Server	92
6.1.3	Requirements for the Security Architecture	94
6.2	Security Mechanisms	95
6.2.1	Protection Layer 4: TPM and Hardware	97
6.2.2	Protection Layer 3: Hypervisor and Management VM	97
6.2.3	Protection Layer 2: Open VM and Trusted VM	98
6.2.4	Protection Layer 1: User interface	99
6.2.5	Attestation Protocol	99
6.3	Evaluation of the Security Mechanisms	101
7	Implementation	103
7.1	History	103
7.2	Components	105
7.2.1	User Interface	106
7.2.2	Copy DB	108
7.2.3	Rule DB	109
7.2.4	Policy Enforcement Point	109
7.2.5	Policy Decision Point	110
7.3	Configuration	110
7.4	Performance Evaluation	110
7.4.1	Performance of Individual Functions	112
7.4.2	Performance of the Creation of Views	114
8	Related Work	115
8.1	Server-side Access Control	116
8.1.1	“Secure and Selective Dissemination of XML Documents”	116
8.1.2	“X-GTRBAC: An XML-Based Policy Specification Framework and Architecture for Enterprise-Wide Access Control”	117
8.2	Client-side Access Control	118
8.2.1	Digital Rights Management	118
8.2.2	Detecting a Compromised System State	119
8.2.3	Integrity Reporting	119
8.3	Usage Control	120
8.3.1	“Relevancy-based Access Control”	121

8.3.2	“Controlling Access to Documents: A Formal Access Control Model”	123
9	Conclusions and Future Work	127
9.1	Conclusions	127
9.2	Future Work	129
	Index	131
	Bibliography	134

List of Figures

1.1	Loss in dollar for different types of incidents (source [GLLR06])	2
1.2	Alternative sets when the desired set cannot be specified . . .	5
2.1	Example of an access matrix	14
2.2	Components of the RBAC model	16
2.3	Example of a role hierarchy	17
2.4	An example of the tree organization of the objects	19
2.5	Situation after access to objects o_6 and o_9	20
2.6	XML example document	22
2.7	Example usage of Namespaces	24
2.8	Example of a Schema for a Report	25
2.9	Integrity reporting protocol [SZJvD04]	29
3.1	Role hierarchy of the scenario	32
3.2	Data transfers of Situations 1, 2 and 3	33
3.3	Sets of objects with different permissions	35
4.1	Illustration of the <i>is-copy-of</i> relation	49
4.2	Syntax of access control rules	50
4.3	Rule denying view	55
4.4	Rule denying view	56
4.5	Rule denying deleting of a section	57
4.6	Additional functions	59
4.7	An example of the objects in the CWM	59
4.8	Rule enforcing the read policy	60
4.9	Strict rule enforcing the write policy	61
4.10	More flexible rule for the write policy	61
5.1	System architecture	67
5.2	Protocol steps for performing an operation in Combination 2 .	81
5.3	Protocol steps for performing an operation in Combination 4 .	83
5.4	Distributed System architecture	84

6.1	Security architecture organized in layers	96
6.2	Attestation protocol	100
7.1	Example of a history element	105
7.2	Components implemented in the prototype	106
7.3	Screenshot of the User Interface	107
7.4	Example of the XML representation of the copy database . . .	108
7.5	Example of a policy document	109
7.6	Screenshot of the Server Configurator	111
8.1	Sticky policies [SBO06]	124

List of Tables

3.1	Different protection levels of documents in the scenario	32
3.2	Document types used in the scenario	32
4.1	Versions of the create operation and their parameters	45
4.2	Versions of the delete operation and their parameters	45
4.3	Versions of the copy operation and their parameters	46
4.4	Parameters of the change-attribute operation	47
4.5	Versions of the view operation and their parameters	48
4.6	Getting the copies of an object	54
4.7	Getting related nodes depending on time	56
4.8	Getting the context of a history entry	57
4.9	Getting accessed nodes	58
4.10	Getting specific nodes of a rule	58
5.1	Components of our system architecture	79
5.2	Possible sides to implement the components	80
5.3	Comparison of the four combinations	84
6.1	Attacks on the client and required mechanisms	91
6.2	Attacks on the server and required mechanisms	94
7.1	Arguments for storing operations in histories	104
7.2	Specification of the test system	112
7.3	Summary of the performance of different groups	113

List of Algorithms

1	Create View	69
2	Evaluate Rules	70
3	Create Element	71
4	Create Text Content	71
5	Split Text Block	72
6	Copy Element	73
7	Copy Text Content	73
8	Delete Text Content	74

Chapter 1

Introduction

In this chapter, we will give a motivation for access control in general and history-based access control in particular. In addition to this, we give an outline for this thesis.

1.1 Motivation

In the recent years, there was a development in which paper documents were more and more replaced by electronic documents. As electronic documents introduce several advantages, as space saving storage, faster electronic transfer and the possibility to perform an electronic search, they also bring new risks. One such risk is that electronic documents can be stolen more easily. For example, a hacker can steal electronic documents remotely without entering the building where the computer on which the documents are stored is located. According to studies of the FBI/CSI [GLLR05, GLLR06], unauthorized access and information theft are responsible for a major part of damages caused by computer crime. In addition to that, electronic documents can be more easily destroyed, e.g., one inconsiderate command can delete an entire document. Moreover, it is much more difficult to control the propagation of information contained in electronic documents, since they can be copied very efficiently compared to paper documents. As a consequence, criminals exploit the new opportunities of electronic documents to steal digital information.

Figure 1.1 shows the amount of loss for several types of computer crime incidents as reported in [GLLR06]. The second largest amount of loss is attributed to unauthorized access to information, which clearly shows the high demand for access control mechanisms, which are mechanisms to restrict access to authorized persons. In other words, these mechanisms define who is allowed or denied to access which object. The fourth largest amount of

loss is caused by the theft of proprietary information. This sum of this type of loss and of unauthorized access exceeds the amount of loss by virus contamination. Since the sketched types of loss can be reduced with access control mechanisms, this highlights the importance of access control even more. Moreover, Figure 1.1 shows that only a relative small amount of loss is created by outsiders penetrating the system. This indicates that protection mechanisms should focus on inside attackers, which are authorized users of the system.

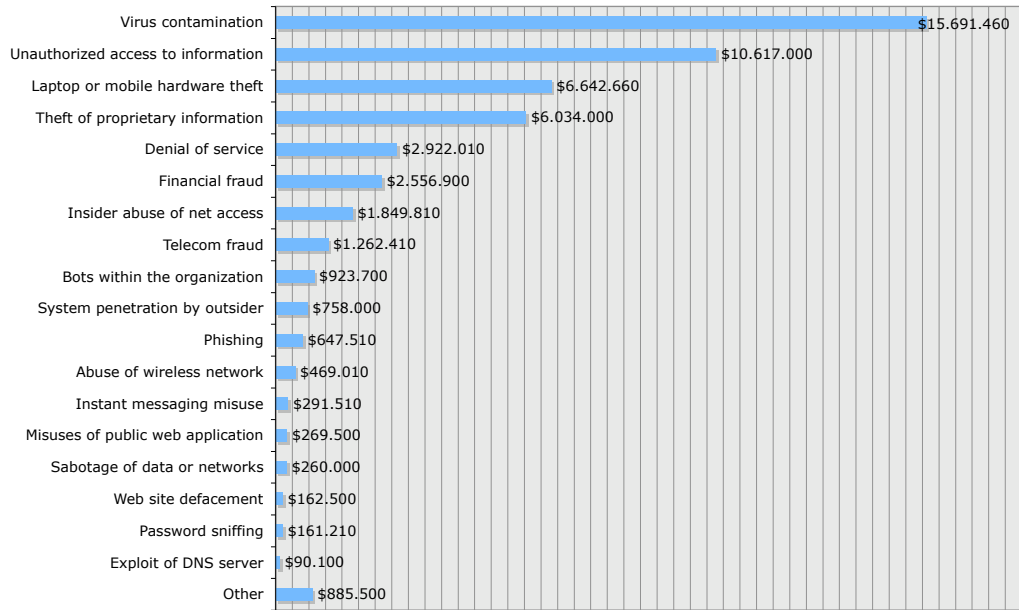


Figure 1.1: Loss in dollar for different types of incidents (source [GLLR06])

Access control mechanisms were introduced, to reduce the risks described above, namely the risk of information theft and the risk of accidental deletion. Many different solutions were developed to provide access control for different types of storing data. Access control systems must be designed differently for different methods of storing data, because the methods can differ significantly in structure and in semantics. For example, concerning XML documents, it is not useful to grant access to a node without granting access to the parent node, because nodes in an XML document have a parent child relation and they form a semantical unit. Without the parent node, the semantics of the child node is not clear. In contrast to this, the semantics of files stored in directories is much different. Here, granting access to a subdirectory without the parent directory is acceptable. Moreover, tables in a relational database have different semantics than XML documents and files in a file system. In

addition to this, different representations of data can require a different level of granularity. For example, for a system that stores files of different formats and the internal structure of these files is not known, it is useful to define access on the level of files. In contrast to this, a system that stores files of one specific type and the internal structure of these files is known, it can be required to define access to parts of these files individually. In this case, it also depends on the structure of these files how access must be defined, e.g., it makes a big difference whether these files contain flat list structures or the data is organized as a tree. As a consequence, we will have a closer look on methods of storing data and different document formats.

Up to then, many different data formats and methods to store data existed. Most applications had their own data format and the interoperability between systems with a different data format was limited. Moreover, some systems store their data in databases, whereas others use files organized in a specific directory structure. To exchange data the corresponding formats must be converted from one format to the other. In some cases, this was especially difficult, since some formats were highly dependent on the properties of the processing computer, e.g., the byte order of the processing computer. Besides these problems, some formats suffered from the problem, that they were difficult to extend. More simply, some formats were designed in a way, that does not allow to store information that was not specified in the existing format. For example, some formats stored the date as two decimal digits, which caused problems when the date changed from the year 1999 to 2000.

As a result of the previous considerations, the Extensible Markup Language (XML) [Con04] was introduced in 1998. The language XML offers several advantages compared to other representations of data, e.g., binary representation. First of all, it is both readable for humans and for computers. Moreover, it supports the storing of unicode text and can represent common data structures like lists and trees.

Although much work on access control in the areas of file systems or relational databases has already been done, defining access to XML documents is a different issue as stated in [FM04]. The structure of XML documents is not always known in advance, e.g., if an XML document has no schema. Moreover, as stated above, elements in XML documents are much more dependent on their context than files in a file system or records in a database are, e.g., it is not always useful to grant access to an element without granting access to its descendants.

Since XML is widely used and offers many technical advantages, we focus on access control for XML documents in this thesis. As a remaining question, we must evaluate how we want to define access for XML documents. There are many different approaches for this task, which we will discuss in the

following.

Previous methods for access control defined access for each object individually and required to manually maintain lists of allowed and denied objects. For example, newly created objects must be manually added to the access control list. But this approach has several drawbacks. First, the manual assignment of access rights to objects is both inefficient and error-prone. Additionally, this approach does not allow to implement a central access control strategy in an automatic and systematic way. Instead, we need an approach in which objects are described by their properties and access control definitions are derived by rules which make use of these properties. In contrast to the manual approach, only the set of rules must be maintained and it must only be updated when the overall access control strategy changes. This rule-based method of specifying access is required in many business scenarios due its advantages in security and its lower effort of maintenance.

In these scenarios, company internal rules define how access should be granted. For example, a set of documents to which access should be denied can be defined by a set of rules that specifies the corresponding documents by using conditions on certain properties of these documents. We refer to these rules as access control rules. When new documents are created or existing documents are changed the access control rules are evaluated again and the resulting set of documents and their corresponding permissions is updated. To sum up, instead of manually maintaining access control lists, access control rules can be used to define access with the help of conditions about certain properties of the objects automatically.

Models for access control for XML documents differ in their expressiveness for defining the conditions used in access control rules. The models expressiveness for these conditions has direct impact on both usability and security. In the following, we will discuss the consequences of having an access control model that lacks expressiveness and cannot express the required conditions to express access control rules for a specific scenario.

If the used model is not able to express a required condition for granting or denying access in an access control rule, an alternative condition must be chosen that can be expressed with the model. There are three different types of alternative conditions besides the intended condition. Figure 1.2 illustrates the resulting sets of objects, depending on the chosen alternative condition.

The first alternative is to chose an expressible condition that specifies a set of objects which is too large and includes the intended set of objects. If the access to these objects is denied, access to some objects is denied even though it was not intended to do so. The result is a limitation of usability, because access is denied in a case where it is not required. If the access to

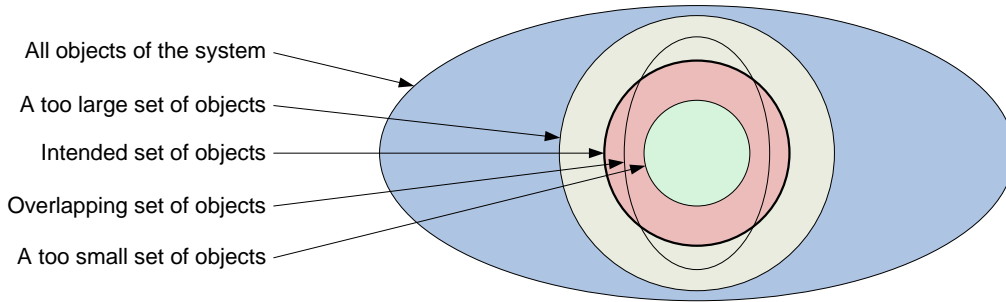


Figure 1.2: Alternative sets when the desired set cannot be specified

these objects is allowed, access to objects is granted, which is not intended to be granted, which results in a security problem, since access to some objects is granted but should be denied.

The second alternative is to choose an expressible condition that both misses some objects that should be included and selects additional objects that should not be included in the resulting set. The result of such a condition is a set which overlaps with the intended set of objects. In any case, such a specification both limits the security and the usability.

Finally, the third alternative is to choose an expressible condition that specifies a set that is a subset of the intended set of objects. This alternative is the opposite to the first alternative. Depending on whether the access to these objects is allowed or denied, this specification limits either security or usability.

As a result of the previous discussion, we learn that the expressiveness of the model for defining access control rules is very important. However, the expressiveness always depends on the requirements of a specific scenario, because different scenarios can require very different conditions for specifying objects in access control rules. As a consequence, there is not absolute measurement for expressiveness. We only can say that the expressiveness of a model for specifying objects in access control rules is sufficient for a specific scenario.

In the previous discussion, we focussed on specifying objects in access control rules, but the concept of defining objects by rules can also be used for subjects. In many cases, there are much more objects than there are subjects in the system. For this reason and to reduce complexity, we focus on how to specify objects in this thesis. Nevertheless, approaches that focus on the specification of subjects can be combined with our approach as these two aspects are independent of each other.

The question is what are the properties that should be used in access

control to define access. A common approach is to use the content of a XML document itself in access control rules to define access. An XML document consists of elements that are organized in a tree structure. These elements have a name and can optionally have text content and attributes that further specify the element.

We can use the tree structure of the XML documents to define subtrees which are allowed or denied to access. Consider a set of equally structured XML documents, which have a root element `Patent application`. This root element has two children elements, namely `Description` and `Main part`. The first element contains a public description of the patent application, whereas the second elements contains the confidential content of the patent application. Both elements can have children elements themselves, which contain the actual content of these parts. In this example, we could allow access to the subtree starting at the element `Description` and deny access to the subtree of `Main part`. In addition to this, we can specify objects more precisely if we also use attribute values in conditions. In addition to the previous example, assume that the `Patent application` has an attribute `granted`, which is either set to `true` or `false`. If the attribute is set to `true`, the patent application is granted and no longer confidential, otherwise it is not granted and still confidential. Next, we can formulate access control rules that use attribute values. Thus, we can now specify that access to the main part of a patent application is only denied when the corresponding attribute `granted` is set to `false`. In this example, we raise usability since, we remove the unnecessary restriction that granted patent application cannot be accessed.

Summing up, the content of an XML document is a very useful aspect to be used in access control rules. As a consequence, many approaches [BF02, DdVPS02, MTK03, GB02] define access to parts of XML document depending on the content of the document itself, e.g., attribute values, tag names or text content. Although the approach of using the content allows to express complex conditions on the content of an XML document, certain policies cannot be expressed. This regards all policies that depend on information which is not contained in the XML document. This information is referred to as *context information* and is used to describe the situation or the document parts more precisely. Common types of context information are location and time, e.g., in [BGBJ05] access can be defined depending on time, the model described in [BCDP05] allows to define access depending on location, whereas [Han07] considers both time, location and additional aspects. With that information access to certain documents can be restricted to specific locations or specific times, e.g., sensitive documents can only be accessed within the company building and within the office hours. Besides

time and location there are many other kinds context information, e.g., the team to which the current user [Tho97] belongs or the situation [BEE07], which is defined by properties of the network connection and the proximity of other detectable devices. Many, kinds of context information require sensor hardware to capture the context information. For example, the location of human users can be measured using the Global Positioning System [PS96] and nearby persons can be detected with cameras and image recognition.

All the context information described above helps to characterize the situation of the access or the accessed object more precisely. These characteristics can be used in access control rules to define more precisely what is allowed and denied. Some of this context information is more useful and other context information can be less useful for security or usability. For example, the room temperature, which is a specific context information, is not relevant for most of the scenarios.

However, there is another type of context information that can be very helpful for both security and usability. We think that the information about how a document was created is very useful for access control, because the way how the content was created can contain many important aspects, which can be used in access control rules. We refer to all information concerning how a document was created as the *history*. We will give two motivating examples of aspects of the history, which can be important for access control.

Reconsider the example with the patent application from above. There, we used the location of a document part within a document to define access to it. In today's business world the concept of reusing existing information is very important, because it is inefficient to recreate document parts that have been already created elsewhere. Therefore, document parts are copied between different documents. As a consequence, in addition to the current location of a document part, it is also very helpful to know from where it was copied, in case it was not created from scratch. For example, assume that a part of the main part of the patent application is copied to another document. In that case, the current location of that copied document part is not so relevant, but it is very important to know from where it was copied. It would be very useful to define an access control rule that states “deny access to all document parts that are copied from the main part of a pending patent application”.

The history can contain many useful aspects that help to specify objects in access control rules. Another such aspect is the knowledge of who has created a specific document part. In addition to the knowledge which individual person has created a specific part, it can be helpful to know in which job function the subject was active in. This knowledge helps to further characterize a document part. In some cases, it might not be relevant

where a specific document part is located, but by whom it was created or modified. For example, assume that some documents are created by subjects with differently ranked job positions, e.g, **junior researcher** and **senior researcher**. In this example, junior researchers develop suggestions and senior researchers make the final modifications. In this example, it is required that after a senior researcher has modified a part of a document, the junior researches must be denied to make further modifications. Similar processes occur in many scenarios, where subjects in a higher job position have the authority to declare something as final.

Above, we illustrated that history information is important and helpful for access control. In addition to the content of a document, the information how this content was created helps to determine its protection requirements and finally to define access to it. To the authors knowledge there is no model aiming at history-based access control. Therefore, the goal of this thesis is to develop such a model. Due to the wide usage and importance of XML, we will develop a model for history based access control for XML documents. Within this thesis, we will also discuss which parts of the history must be considered and how they can be used within access control.

1.2 Structure of this Thesis

The remainder of this thesis is organized as follows: In Chapter 2, we provide background information about three technologies used in this thesis. The first of these technologies is access control, which is the main topic of this thesis. We explain the basic concepts of access control and give an overview of common models for access control. The second technology is the extensible markup language (XML), which is the format of the documents on that we focus in this thesis. We present details about XML itself as well as information on related technologies. The third technology is about the concepts defined by the Trusted Computing Group, which we use as a part of our security architecture. In Chapter 3, we present a scenario to illustrate several challenges for an access control model for XML documents. Next, we extract several individual requirements for our access control model from these challenges. After this, we describe our model for access control for XML documents that we have developed based on the requirements in Chapter 4. Since, we have specified our model on an abstract level, we must design a system architecture that supports our model in Chapter 5 to apply the model in a real world scenario. In Chapter 6, we first perform a risk analysis for our previously defined architecture and then describe security mechanisms that reduce that kind of risks. After this, we give details on the implementation of

our system architecture in Chapter 7. We use this prototype to demonstrate feasibility of our approach. Moreover, we present the result of a performance evaluation of the implementation. Next, we present related work in three different areas in Chapter 8. We explain previous work on access control, on enforcement mechanisms and on integrity reporting. Finally, we conclude and point out future work in Chapter 9.

Chapter 2

Background

In this chapter, we provide background information for the technologies used in this thesis. In Section 2.1, we give an introduction to access control and present some common models which are helpful to understand our model presented in Chapter 4. Next, in Section 2.2, we give details to the Extensible Markup Language (XML), which we use to represent the documents and the corresponding metadata of our model in Chapter 4. We present information to the XML Path Language (XPath) in Section 2.2.5, which is the language to define the applicable objects of the access control rules of our model. Finally, in Section 2.3, we present details on the mechanisms defined by the Trusted Computing Group (TCG), which we use as part of our security architecture in Chapter 6.

2.1 Access Control

In this section, we give an introduction to access control. For this purpose we first explain access control in general and then we continue with the description of some common models for access control.

The purpose of access control is to limit the actions or operations that a legitimate user of a computer system can perform. [SS94]

Subjects are the more general concept of users to describe the active components of a system. A subject can either be a human user, a process that acts own behalf of a human user or an autonomous process acting on its on behalf, e.g., a software agent or a maintenance process of the operating system.

Subject perform their operations on objects, which are the passive components of a system and can be any kind of resource of the system, e.g., a file,

a network socket, a data base entry or a variable in a program. Objects can be modeled in different granularities. This granularity defines what is the smallest data unit an object can represent. For example, if a directory is the smallest unit an object can represent, we regard that as a coarse granularity. In contrast to this, if parts of files are treated as the objects of the system, we call this a fine granularity.

Depending on the kind of objects, there can be different modes of access. A mode describes what kind of access is performed. Common modes of access are read, write and execute. But if the object is a variable in a program, possible access modes can be update, decrement or increment. In many systems, access modes are also referred to as operations or as access attributes.

Independent of the way subjects, objects and access modes are modeled, access can be defined in three different ways. One option is to define what is allowed and to assume that access to not specified objects is denied. This concept is referred to as having positive permissions. The next option is to define what is denied and assume that the not specified objects are allowed to be accessed, which is referred to as having negative permissions. The third option is to combine both concepts and define both what is allowed and what is denied. Conflicts can occur if a specific access is both allowed and denied. To solve this problem, a conflict resolution strategy must be defined. The simplest of such strategies is to define that either deny overrides allow or vice versa. More complicated strategies use priorities to define which statement takes precedence.

Moreover, there are two different overall strategies for access control models in which way access is defined. The first strategy is *Discretionary Access Control* (DAC), which is defined by the *Trusted Computer System Evaluation Criteria* (TCSEC) [Uni85] as follows:

A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Systems or models that use DAC have in common that permissions are defined for individual objects and individual subjects. The definition is usually performed manually. In most cases, the owner of an object defines these permissions. In most cases, the owner is the subject that has created the corresponding object. As a consequence there is only one owner. The owner

also defines permissions of other subjects on his objects. In [Uni85], the concept of an owner is not mentioned. Therefore, DAC does not necessarily imply that permissions are defined by the owner of an object. Although, most DAC system do so. This is why the term DAC and the owner concept are often used synonymously.

Systems based on DAC have the disadvantage that permissions are not necessarily defined in a systematical way. Each subject may apply its own strategy and it is in the responsibility of that subject that these permissions are set up correctly, which means that only subjects which are supposed to access an object actually have the permissions to do so. While DAC is convenient in scenarios, where users manage objects, for which they are the only person that is responsible, e.g., a user managing his private files on his home computer or a self-employed individual editing his own business documents. In scenarios, where users manage objects on which they are legally restricted in their usage, e.g., employees of a an enterprise handling confidential enterprise documents, DAC is not an appropriate choice. In these scenarios, it is more advisable to use MAC, which we describe next.

The opposed strategy to DAC is MAC, which is short for *Mandatory Access Control*. MAC is defined in [Uni85] as follows:

A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.

In systems using MAC, permissions are derived from the properties of an object with the use of rules. These properties can be represented by a label that is attached as meta data to the object or they can be properties of the object itself, i.e., its content. Subjects are treated in the same manner. Instead of defining or deriving permissions for individual subjects, subjects are grouped by their properties, e.g., by their clearance or by their function within the organization. It depends on the system whether the subjects still have limited options to define permissions. Im some models, the least restrictive permissions are defined automatically by the system and the subject has the choice to make permissions more restrictive.

Next, we continue with the description of some common models for access control. We start with the most basic and oldest model, which is the access matrix model [Lam71, GD72]. This model is also a a basis for the model described in Section 2.1.3.

2.1.1 Access Matrix Model

In the *Access Matrix model* [Lam69, Lam71, GD72], access is defined with the help of a matrix. Since this matrix can change over time, we denote it with the index t for a specific point in time as M_t . The objects O_t , for which access should be defined, define the columns of the matrix. Analogously, the subjects S_t define the rows of the matrix. The resulting matrix has the dimensions $|S_t| \times |O_t|$. Each entry of the matrix, denoted as $M_t(s, o) = \{r_1, \dots, r_n\}$, defines the access rights of the subject s for the object o .

An example of such a matrix is illustrated in Figure 2.1. The illustrated matrix defines the access rights of three users to four different files. In this case, **User 1** is allowed to read files 1, 2 and 4 and to write file 1. **User 2** is allowed to read files 2 and 3 and to write file 1. Finally, **User 3** is allowed to read files 2, 3 and 4 and to write files 3 and 4.

	File 1	File 2	File 3	File 4
User 1	write, read	read		read
User 2		write, read	read	
User 3		read	write, read	write, read

Figure 2.1: Example of an access matrix

There are two different types of models using access matrices: Models with a access matrix, where the matrix can change over time and models with a matrix, where the initially defined matrix does not change. In the first case, additional operations are used to modify the matrix. In [HRU76], Harrison, Ruzzo and and Ullman define six *primitive operations* for that purpose. The operation **enter** adds new rights, whereas **delete** removes rights. The operations **create subject** and **delete subject** are used to create and delete subjects. Similarly, the operations **create object** and **delete object** are used to create and delete objects.

While the Access Matrix model is a simple model, it still offers the possibility to model subjects and objects with an arbitrary fine granularity. If the implementation of the model stores the matrix as a two-dimensional array the amount of required memory will be high, especially when the number of subjects and objects is high. To solve this problem, efficient implementations of this model store the rows or the columns of the matrix individually. If the matrix is stored as a set of rows the implementation is referred as a capability-based system. In the other case, the implementation is described as an access control list. In both cases, the cells are labeled to denote to which subject or object they refer to and empty entries can be omitted to save resources. In situations with a high number of subjects and objects, the

amount of saving can be high, because subjects often have access only to a fraction of the total number of objects of the system.

On the downside, the model does not have a high level of abstraction, because all subjects and objects are treated equally, have no further properties and are not related to each other. For example, two subjects that perform the same task and therefore need the same access rights require manual steps to assign the same access rights to both of them. As a consequence, the costs to update and maintain the access matrix can be high. Moreover, if there is a high number of subjects and objects the matrix will be big and manual maintenance of the matrix can become difficult. In the following section, we will discuss a model which offers a higher level of abstraction. As a consequence, this model simplifies the maintenance.

2.1.2 Role-Based Access Control (RBAC)

The concept of Role-Based Access Control (RBAC) is described in [SCFY96] by Sandhu et al. in 1996. Since our model, which we describe in Chapter 4, uses roles to model subjects, we describe the RBAC model in this section.

Instead of assigning access rights, which are called permissions in RBAC terminology, directly to subjects, permissions are assigned to roles. Such a role is used to model a specific task. Therefore, all permissions that are required to perform a specific task are assigned to a role, e.g., a role “cashier” can be used to collect all permissions that are required for performing the job of a cashier. In addition to assigning permissions to roles, RBAC assigns subjects to roles. A role must be activated by a subject to use the corresponding permissions. Multiple active roles of a subject are referred to as a *session*. A subject can also have more than one session. The relations between the components of the RBAC model are illustrated in Figure 2.2. An RBAC model consists of the following components:

1. U , the set of users of the systems,
2. R , the set of roles defined in the system,
3. P , the set of all permissions of the system,
4. S , the set of sessions,
5. $PA \subseteq P \times R$, the relation which assigns permissions to roles,
6. $UA \subseteq U \times R$, the relation which assigns roles to users,
7. $user : S \rightarrow U$, the function which maps each sessions to a single user,
and
8. $roles : S \rightarrow 2^R$, the function which maps each sessions to a set of roles.

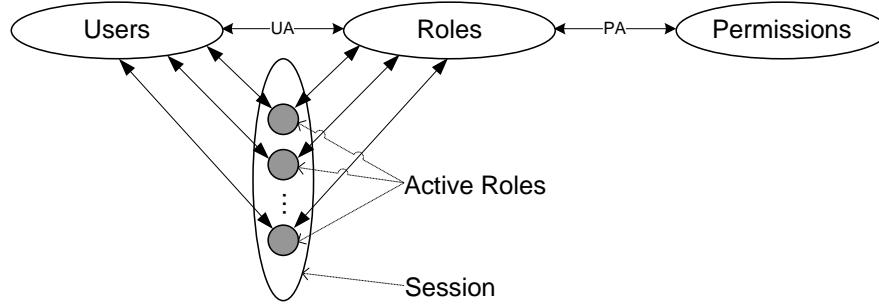


Figure 2.2: Components of the RBAC model

In addition to this base model for RBAC, other concepts are described in [SCFY96] that increase the flexibility and expressiveness of the model. The first such concept are role hierarchies, which define a partial order among the roles of the system. This partial order defines a specialization relationship among the roles, where the more special role inherits all permission from the less special role. As a consequence, more special rules have more permission than less special rules.

Figure 2.3 shows an example of such a role hierarchy. By convention less powerful rules are depicted at the bottom and more powerful rules above them. In this example, **Project Member** is the role with the smallest number of permissions. Both, **Engineer** and **Researcher** are specializations of this role, inherit its permissions and add further permissions to it. **Senior Engineer** and **Senior Researcher** both inherit permissions from their corresponding non-senior roles. Finally, the role **Project Supervisor** is derived by multiple inheritance from both senior roles. As a result, this role combines the permissions of both senior roles.

In general, multiple inheritance can lead to problems, where the resulting role inherits more privileges than are required to fulfill the corresponding task, which violates the *principle of least privilege* [SS75, Den76]. Saltzer et al. describe this principle in [SS75] as follows:

Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

This principle aims at reducing misuse of privileges, e.g., fraud, and at reducing errors where privileges are used accidentally in a wrong way, e.g., accidentally deleting an important file.

In [SCFY96], Sandhu et al. describe *constraints* as another mechanism that can be added to RBAC. Constraints can be defined both for role to user assignment and for role activation. The first case is used to model a static

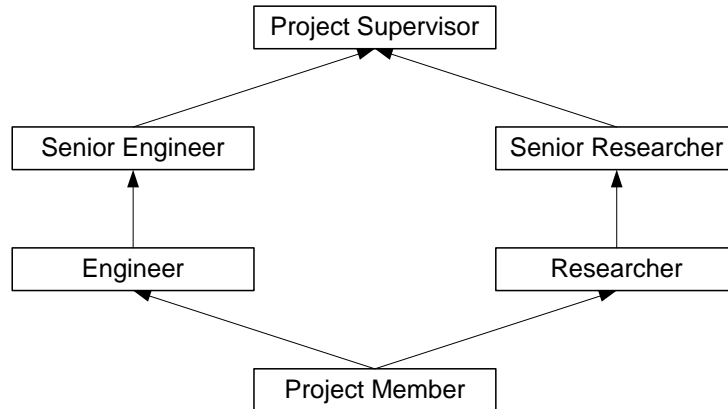


Figure 2.3: Example of a role hierarchy

separation of duties, which defines that certain roles are mutually exclusive. As a consequence, a user can be assigned only to one role of a set of mutually exclusive roles. This mechanism is designed to prevent fraud in situations where a user is able to act in two or more roles for certain tasks that should not be executed by a single person, e.g., chief buyer and accounts manager are two jobs that should not be performed by a single person, even if they are not executed at the same time.

Static separation of duty is too strict for some cases, where it is only required that different roles are not activate by one user at the same time. For this case, constraints are defined for role activation to model dynamic separations of duties. Here, sets of roles are defined that can not be activated simultaneously. This mechanisms is designed to prevent fraud, where a user would need two different permissions simultaneously to commit fraud, e.g., an employer of a bank can also be as costumer of the bank, but he should not be active in both of the corresponding roles at the same time.

In addition to separation of duties, other constraints can be defined as well. For example, *cardinality constraints* define that a specific role has a maximum number of members, e.g., there should be at maximum one project supervisor. Other constraints can be defined for the assignment of permissions to roles, which can be used to ensure that certain powerful permissions are only assigned to a small number of roles.

In addition to the constraints discussed so far, many other types of constraints are described in literature. For example, in [BBF01] Bertino et al. describe activation constraints for roles that depend on the current time and the time of events that occurred in the system. In [AC04], Alotaiby et al. present constraints that depend on the configuration of teams. The activation

of roles depends on the geographical position of the subjects in [BCDP05].

Moreover, there are many other models which are based on RBAC. In [CMA00], Convington et al. specify “Generalized Role-Based Access Control” (GRBAC) which extends RBAC in two aspects. *Object roles* are used similarly to the subject roles of RBAC. They classify objects by their properties, e.g., level of confidence or document type. *Environment roles* classify the environment by properties which are considered to be relevant for security. These Environment roles can be used as an activation constraint for subject roles.

In [KKC02], Kumar et al. describe a similar but more flexible approach. In this approach, an *object context* contains all security-relevant properties of an object. Similarly, an *Context/user* contains the security-relevant properties of a user. A *role context* defines a condition for activating a specific role and is a boolean expression, which uses attributes of the object context and of the user context. As a result, both objects and users can be described in a flexible way independent of concrete instances by classifying both by their properties. Moreover, the definition of roles can be based on the properties of the user and the object.

2.1.3 Chinese Wall model

The *Chinese Wall model* [BN89] was designed by Brewer and Nash to prevent misuse of insider knowledge. When the Chinese Wall model was introduced, this problem was most critical in the financial domain, e.g., a bank with customers that are competitors to each other. In such cases, a bank employee should only have access to the objects of one of the rivaling enterprises. A similar scenario is the consulting business. Here, a consultant should have access to only one company out of a group of companies in the same business. Today, the problem of not accessing documents of competing companies affects the entire service sector. Since we use the Chinese Wall model as a motivating example in Chapter 4, we briefly describe it in this chapter.

The model is based on an access matrix (see Section 2.1.1) and defines three modes of access: read, write and execute. The objects of the model are organized as a tree structure with three levels and the objects being the leafs. The level directly above the objects is the level containing company data sets, where each node represents a different company. All objects attached to one such node belong to the corresponding company. The level directly above the company data sets denotes the *conflict-of-interest classes*, where each node represents one such class. The children of such a node are companies that are competitors to each other.

An example of such a tree is depicted in Figure 2.4. In this example,

we have two conflict-of-interest classes, which are “Banks” and “Petroleum Companies”. The class “Banks” consists of “Bank A”, “Bank B” and “Bank C”. Similarly, the class “Petroleum Companies” consists of “Oil Company A” and “Oil Company B”. The children of a company node are the objects that belong to the corresponding company, e.g., the objects o_8 to o_{10} belong to “Oil Company A”.

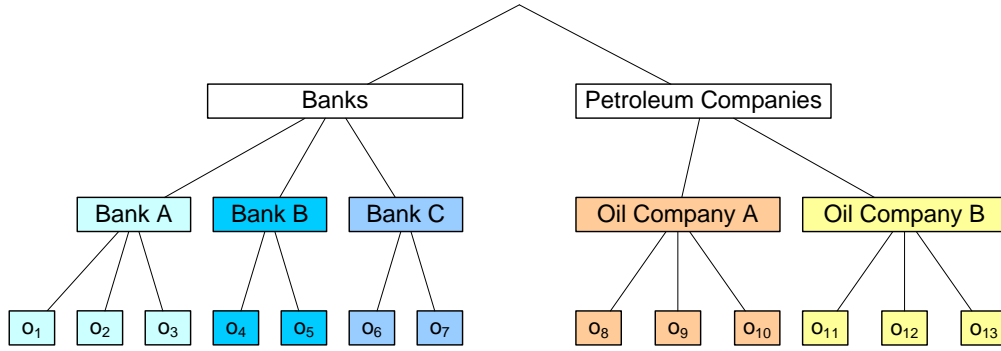
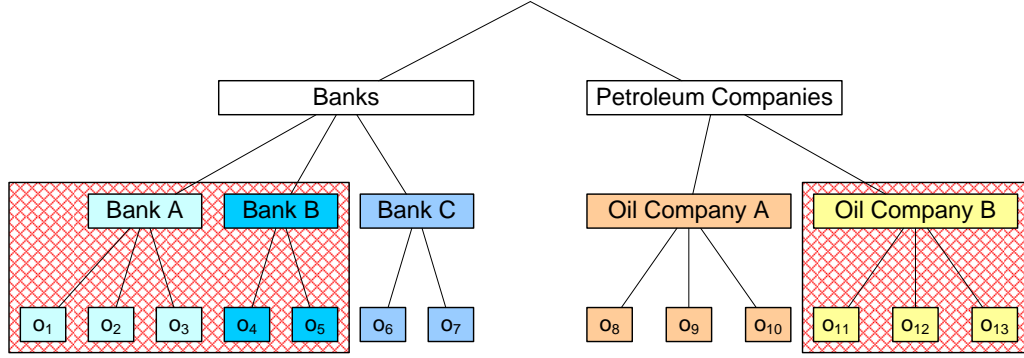


Figure 2.4: An example of the tree organization of the objects

As illustrated in the previous example, each object belongs both to a company data set and to a conflict-of-interest class. In the model, x_i denotes the the conflict-of-interest class of the i -th object o_i and y_i denotes the corresponding company data set. In our example, x_5 is “Bank B” and y_{12} is “Petroleum Companies”. A special case is denoted by y_0 and x_0 , which is used for public information that is not restricted in any way.

The Chinese-Wall model uses two matrices to define access. The first matrix M defines the permissions that subjects have on different objects. This matrix works as described in Section 2.1.1. In addition to this, a second matrix N records the history of access. The matrix N has the dimensions $|S| \times |O|$, where S is the set of subjects and O is the set of objects of the system. Each cell of this matrix corresponds to a pair of subject and object and denotes whether the subject already had accessed the corresponding object. The cell $N(i, j)$ contains the access modes that the i -th subject had used to access the j -th object. If the object was not accessed by that subject, the cell is empty. After each access, the matrix M is updated to reflect to new situation.

The Chinese Wall model uses two rules to define access. The first rule is used for read and execute access and states that access to an object o_i is only granted if the accessing subject did not access any object of the same conflict-of-interest class that belongs to a different company. As a consequence, after an access to one object, all objects of the same class that

Figure 2.5: Situation after access to objects o_6 and o_9

belong to different company data sets are blocked for that subject. This rule can also be expressed formally. A subject s_i is allowed to perform an access $a \in \{read, execute\}$ on the object o_j if and only if:

$$a \in M(s_i, o_j) \wedge \forall o_k \in O, k \neq j : \\ N(s_i, o_k) \neq \emptyset \implies (y_k = y_j \vee x_k \neq x_j \vee y_j = y_0).$$

This definition states that a subject s_i first of all must have the required privilege, which is formally expressed as $a \in M(s_i, o_j)$. For all other objects that have been accessed so far ($N(s_i, o_k) \neq \emptyset$), at least one of the following three conditions must be true. First, the object belongs to the same company data set, which is formally expressed as $y_k = y_j$. Second, the object is in a different conflict-of-interest class, i.e., $x_k \neq x_j$. Third, the object is marked as being public information, which is formally expressed as $y_j = y_0$.

Figure 2.5 illustrates the rule presented above and shows the situation after a subject has accessed objects o_6 and o_9 . In this case, objects from “Bank C” and “Oil Company A” have been accessed. As a consequence, access to objects of “Bank A”, “Bank B” and “Oil Company B” is denied.

This rule alone is insufficient, because unwanted data transfer that enables misuse of insider knowledge is still possible. For example, the subject s_1 reads o_1 and writes the corresponding data to o_8 . After that, the subject s_2 reads that data from o_8 and writes it to o_4 , where it should never be written too, because it belongs to an competitor of “Bank A” to which o_1 belongs to. As a consequence, another rule is required that prevents this type of data transfer. A subject s_i is allowed to write to an object o_j if and only if:

$$write \in M(s_i, o_j) \wedge \forall o_k \in O, k \neq j : \\ read \in N(s_i, o_k) \implies (y_k = y_j \vee y_j = y_0).$$

This rule prevents transfers like the one sketched above, because it allows to write data to an object only if the previous read accesses ($read \in N(s_i, o_k)$) refer to the same company data set, i.e., $y_k = y_j$, or the read information is public, i.e., $y_j = y_0$.

The Chinese Wall model allows to model subjects and object with arbitrary fine granularity. Its basis is the matrix M where users define access rights for individual objects. This basic strategy follows the DAC principle, whereas the overall strategy that enforces the chinese wall is a MAC strategy. One drawback of the model is that it is based on an access control list, which is not desirable in an enterprise scenario. Moreover, the model is designed for a special type of scenarios and is limited to enforce only one type of mandatory policies, which is the chinese wall policy. It is not possible to specify a policy that is more specific for the current scenario, e.g., exceptions to the chinese wall policy or more fine-grained rules which also depend on the type of document.

2.2 The Extensible Markup Language

In this Section, we give an introduction to the Extensible Markup Language (XML), explain how XML documents are structured and describe how XML documents are specified with the use of an XML Schema. In addition to this, we describe how document type definitions and schemas can be used to define the structure of a document. After this, we will explain the XML Path language, which is used to address specific parts of an XML document.

2.2.1 Introduction to XML

The *Extensible Markup Language* (XML) [BPSM⁺06] is a standard for a language recommended by the World Wide Web Consortium (W3C) [Wor06] and is used to describe XML documents. We use XML in this thesis as the format of the documents that we want to protect. Therefore, we will present a summary of the technologies related to XML, which are helpful to understand this thesis.

The language XML was designed to exchange data between different computer systems and is a simplified subset of the Standard Generalized Markup Language (SGML) [ISO86]. XML offers five advantages compared to other representations of data, e.g., binary representation. First, XML is both readable by humans and by machines. Second, XML is self-documenting because XML documents both contain a descriptive label for each data element as well as the data element itself. Nonetheless, one has to agree about common

names for labels to enable automated exchange of data. Third, XML documents can contain records, trees and lists, which are common data structures. Fourth, XML documents support Unicode [The91], which is a standard to represent character data and allows XML documents to contain almost any written language of the world. Fifth, the strict and simple syntax of XML documents enables tools to process XML data efficiently. After describing the benefits of XML, we continue with the description of the structure of XML documents.

2.2.2 XML Documents

XML documents consist of elements which can have attributes and further elements as children. Both, elements and attributes, are typed. Elements are denoted as a sequence of *start tag*, *text content* and *end tag*. A start tag is a sequence of “<”, the name of the tag and “>”, e.g., <Report>. An end tag is a sequence of “</” the name of the tag and “>”, e.g., </Report>. Alternatively, elements can also be empty. In this case, elements are denoted in an abbreviated form, which is the sequence “<”, the name of the tag and “/>”, e.g., <Section />.

An example of an XML document is illustrated in Figure 2.6. The first line of this document is the *prolog* of the document. The prolog contains general information about the document, such as the version number of the used XML specification and the document encoding. Moreover, the prolog can contain a schema or document type definition to which the document must be conform, which is not present in this example. The document in Figure 2.6 is represented by its root element **Report**, which has an attribute **funded-by** set to the value **Company A**. The root element has two children. Both of them have the type **Section**. The first **Section** element has both an attribute and text content. The second **Section** element is an example of an empty element.

```
<?xml version="1.0" encoding="UTF-8"?>
<Report funded-by="Company A">
  <Section title="Introduction">
    text of the introduction
  </Section>
  <Section />
</Report>
```

Figure 2.6: XML example document

An XML document must be *well-formed*, which requires it to comply with five conditions. First, an XML document must have only one root element. Second, non-empty elements must be enclosed in both a start and an end tag. Third, all attribute values must be placed either in single quotes (") or in double quotes ("). Forth, tags must be nested and must not overlap each other. In other words, each element except of the root element must be contained completely in another element. Fifth, the XML document must be encoded as stated in its prolog or as specified in the corresponding transport protocol, e.g., HTTP. In addition to being well-formed, a document can also be valid, which means that the document is conform with a document type definition or a schema. We will explain both concepts in Section 2.2.4. Before we introduce these concepts, we must explain the concept

2.2.3 XML Namespaces

XML Namespaces [Con99] are used to combine data from different documents in one XML document. Namespaces can be assigned both to elements and to attributes. This can be done to give elements or attributes with the same name different semantics as it was intended, because Namespaces define in which context an element has to be interpreted. Consequently, the use of Namespaces helps to avoid collisions of data type names from different sources. As a result, documents can be combined even if parts of their data types share the same name but have different semantics.

Namespaces are identified using a Uniform Resource Identifier (URI) [BLFM05], which is the more general concept of a Uniform Resource Locator (URL) [Net02]. These URIs are used to name or to locate a specific resource. Namespaces can be assigned either directly to an element or with the use of a *prefix*. Elements inherit the Namespace of their parent elements, if no Namespace is assigned directly to them. Only the most specific Namespace is effective for an element. Consequently, an element or attribute can have only one Namespace.

Figure 2.7 illustrates the usage of Namespaces. In this example, we assign the Namespace `urn:example:report` to the root element `Report` and define the Namespace `urn:example:ac` with the prefix `ac`, where `ac` is short for access control. We use the prefix `ac` to assign the the corresponding Namespace to the two `block` elements.

In this thesis, Namespaces are used within the implementation in Chapter 7 to store history information together with the original content of a document without having collisions of names of the corresponding elements and attributes.

```

<?xml version="1.0" encoding="UTF-8"?>
<Report xmlns="urn:example:report"
        xmlns:ac="urn:example:ac">
  <Section>
    <ac:block>text part 1</ac:block>
    <ac:block>text part 2</ac:block>
  </Section>
  <Section />
</Report>

```

Figure 2.7: Example usage of Namespaces

2.2.4 XML Schema

XML Schemas [Con04] are used to describe classes of documents with a common structure. Before Schemas were introduced in 2004, Document Type Definitions (DTDs) were used for this purpose. Compared to DTDs, Schemas offer three advantages. First, Schemas are denoted in XML in contrast to DTD, which are written in a different syntax. Storing Schemas as XML enables one to use the same tools for both Schemas and XML documents to verify their well-formedness and their validity. Second, Schemas offer a richer language to express constraints on the structure of documents, which makes the resulting specification more precise. For example, to define a data type in a DTD one has to enumerate all possible values for that type. In a Schema one can use data types or regular expressions for this purpose, which is both more expressive and more precise. Third, Schemas can be derived from each other by inheritance, which makes the definition of Schemas more expressive and therefore more efficient. Schemas can also be combined by using other Schemas in the definition of one Schema. In this cases, Namespaces are used to avoid conflicts in names and to resolve ambiguity.

The Schema in Figure 2.8 defines **Report** documents like the one illustrated in Figure 2.6. In its root element **schema**, the Namespace with the prefix **xs** is defined, which is used for all elements of a Schema. Next, the root element of a **Report** is defined as a complex type (**complexType**) consisting of a **sequence** of **Section** elements and a required attribute **funded-by**. The **Section** elements in the sequence must occur at least 0 times, which states that it is optional, whereas their maximum number of occurrences is unbounded. The **Title** attribute of a **Section** is marked as being optional. Both the attributes and the content of the element are defined as a **string**, which means that they can contain any text.

Summing up, Schemas describe how certain types of XML document must

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Report">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Section">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="Title"
                  type="xs:string" use="optional"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="funded-by"
        type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Figure 2.8: Example of a Schema for a Report

be structured, but they do not define the semantics of the corresponding elements. Schemas are relevant in this thesis in two aspects. First, to define access control rules in our model it is essential that documents conform to the structure defined in their Schemas. Otherwise, it would be impossible to specify access control rules. Second, in the implementation of our model in Chapter 7, we make use of Schemas to define the data structures for access control rules and histories.

2.2.5 XML Path Language (XPath)

The XML Path Language (XPath) [CD99] is a language to address parts of an XML document. XPath is used in many different ways and for different purposes. Many implementations that manipulate XML documents use XPath to specify the objects on which an operation should be performed. Moreover, XPath is also used as a query language to retrieve a specific part of a document that is of special interest. In addition to this, XPath is used in other languages such as XPointer, XSLT and XQuery, which use XPath as a basis mechanism to address parts of an XML document. Finally, XPath is

used in access control to describe the objects within an access control rule. In this thesis, we use XPath as a mechanism to specify objects in access control rules. For this reason, we briefly explain the basics of XPath.

XPath operates on the logical structure of an XML document, which is interpreted as a tree with three types of nodes: elements, attributes and text content. The basic expression of XPath is the path expression, which consists of a sequence of element names divided by “/”, e.g., `/Report/Section` is used to address all **Section** elements of a **Report**. A condition using predicates can be placed after each step in the path and is used to specify the nodes in question more precise, e.g., `/Report/Section[@Title="Introduction"]` only returns the **Section** elements with an attribute **Title** that is set to “Introduction”. These previous examples made use of the child relation of elements among each other. In addition to the child relation, other relations can be used in XPath as well.

An XPath expression is evaluated for a specific node, which is called the *context node*. The context node changes within a sub-expression to the resulting nodes of the previous expression. The result of an expression has one of the following four types: boolean, node set, string or floating point number. XPath defines a set of operators that can be used to manipulate these data types. This includes boolean operators, arithmetic operators and comparison operators. In addition to these operators, XPath has a built-in function library which includes functions for string manipulations, functions on node sets and mathematical functions.

In XPath, other relations of the elements among each other can be used to specify nodes in expressions as well. For this purpose, XPath offers so-called *axis* for each relation. For example, the child axis is used to describe elements by their child relation among each other. The child axis is also the default axis of XPath and is specified the “/” character in expressions. The “/” is part of the abbreviated syntax of XPath, which is used to make expressions shorter. The expanded syntax can be used to make expressions easier to understand or to emphasize a specific relation between nodes. For example, `/child::Report/child::Section` is the expanded syntax of the pattern presented above.

2.3 Trusted Computing Group Mechanisms

This section gives an overview of the mechanisms described by the TCG. For a more detailed description, we refer to the TPM specification [Gro06] or [Pea02].

The core of the TCG mechanisms [Gro06, Pea02] is the Trusted Platform

Module (TPM), which is basically a smartcard soldered on the mainboard of a PC. The TPM serves as the *root of trust*, because its hardware implementation makes it difficult to tamper with, and therefore it is assumed to be trustworthy. One must also assume that the hardware vendor is trustworthy and has designed the TPM chip according to the specification. Although the TPM chip is not specified to be tamper-resistant, it is tamper-evident, meaning that unauthorized manipulations can be detected.

The TPM can create and store cryptographic keys, both symmetric and asymmetric. These keys can either be marked migratable or non-migratable, which is specified when the key is generated. In contrast to non-migratable keys, migratable keys can be transferred to another TPM. Due to its limited storage capacity, the TPM can also store keys on the hard disk. In this case, these keys are encrypted with a non-migratable key, assuring the same level of security as if the keys were stored directly in the TPM. The TPM is able to perform calculations on its own, e.g., it can use the generated keys for encryption and decryption.

In the context of this thesis, the Platform Configuration Registers (PCRs) are of particular interest. These registers are initialized on power up and are used to store the software integrity values. Software components are measured by the TPM and the corresponding hash-value is then written to this platform configuration register by extending the previous value of a specific PCR. The following cryptographic function is used to calculate the values for the specific registers:

$$\text{Extend}(\text{PCR}_N, \text{value}) = \text{SHA1}(\text{PCR}_N || \text{value})$$

For every measured component an event is created and stored in the stored measurement log (SML). The PCR values can then be used together with the SML to attest the platform's state to a remote party. To make sure that these values are authentic, they are signed with a non-migratable TPM signing key, the Attestation Identity Key (AIK). The remote platform can compare these values with reference values to see whether the platform is in a trustworthy state or not. The TCG assumes that a trusted operating system measures the hash value of every process started after the boot process. Such a trusted OS is not part of the TCG specification. For a description of a trusted OS see [GRB03, SZJvD04, Bas06].

The TPM additionally offers a number of different signing keys. One major key is the Endorsement Key (EK) which is generated by the module manufacturer and injected into the TPM. The EK uniquely identifies the TPM and is used to prove that the TPM is genuine. In addition, the EK is used to obtain an Attestation Identity Key (AIK). An AIK is created inside

the TPM, signed with the private portion of the EK, and the public part is transferred to a third party (a Privacy-CA). The Privacy-CA verifies that the platform is a genuine TPM and creates a certificate which binds the identity key to the identity label and generic information about the platform. This certificate, also known as identity credential is sent to the TPM and later used to attest the authenticity of a platform configuration.

2.3.1 Remote Attestation

The remote attestation is used to attest the configuration of an entity to a remote entity. This procedure is widely used to get integrity information before a client proceeds with the communication in order to use a service or receive data, e.g., digital content. This mechanism is referred as integrity reporting and can be applied in many scenarios and different applications, such as controlling access to a network depending on the trustworthiness of the client [SJZvD04]. The integrity reporting mechanism is also one requirement mechanism in the context of DRM applications, since it is obviously required that the DRM-client software is in a trustworthy state and executes a certain policy to prohibit unauthorized use, copy or redistribution of intellectual property [RC05].

2.3.2 Integrity Reporting Protocols

The concept of remote attestation has been developed to enable integrity reporting protocols. In this section we discuss an integrity reporting protocol proposed by [SZJvD04], which is based on the challenge-response authentication [BM92] and is used to validate the integrity of an attesting system.

Figure 2.9 illustrates the remote attestation of B against A. In step 1 and 2, A creates a non-predictable nonce and sends it to the attester B. In step 3a, the attester loads the Attestation Identity Key from the protected storage of the TPM by using the storage root key (SRK). In the next step, the attester performs a *TPM_Quote* command, which is used to sign the selected PCRs and the provided nonce with the private key AIK_{priv} . Additionally, the attester retrieves the stored measurement log (SML). In step 4, the attester sends the response consisting of the signed *Quote*, signed nonce and the SML to A. The attester also delivers the AIK credential which consists of the AIK_{pub} that was signed by a Privacy-CA.

In step 5a, A validates if the AIK credential was signed by a trusted Privacy-CA thus belonging to a genuine TPM. A also verifies whether AIK_{pub} is still valid by checking the certificate revocation list of the trusted issuing

1. A : create a non-predictable 160bit *nonce*
2. A \rightarrow B : ChallengeRequest(*nonce*)
- 3a. B : loadkey(AIK_{priv})
- 3b. B : retrieve $Quote = sig\{PCR, nonce\}_{AIK_{priv}}$
- 3c. B : get stored measurement log (SML)
4. B \rightarrow A : ChallengeResponse(*Quote*, SML) and $cert(AIK_{pub})$
- 5a. A : validate $cert(AIK_{pub})$
- 5b. A : validate $sig\{PCR, nonce\}_{AIK_{priv}}$
- 5c. A : validate *nonce* and *SML* using *PCR*

Figure 2.9: Integrity reporting protocol [SZJvD04]

party. This step was designed to discover masquerading by comparing the unique identification of B with the system identification given in AIK_{pub} .

In the next step, A verifies the signature of the *Quote* and checks the freshness of *Quote* in step 5c. Based on the received stored measurement log and the PCR values A processes the SML and re-computes the received PCR values. If the computed values match the signed aggregate, the SML is valid and untampered. A now only verifies if the delivered integrity reporting values match given reference values, thus A can decide if the remote party is in a trustworthy system state.

Chapter 3

Scenario and Requirements

In this chapter, we first present a scenario to illustrate different challenges for history-based access control for XML documents. Within this scenario, we sketch different situations that illustrate requirements for an access control system. We use these specific situations to derive general challenges for access control. Based on these general challenges, we define explicit individual requirements for our model.

3.1 Scenario

In this section, we present a scenario to illustrate the requirements for history-based access control. In our scenario, we consider five types of documents, which are listed in Table 3.2. These document types differ in their required level of protection. We list protection levels of this scenario and their description in Table 3.1. The document types in this scenario are an example for document types in today's real world scenarios. We believe that documents with similar properties can be found in other scenarios as well.

In this scenario, a project report contains confidential research results, more precisely, the results from a current project. A project summary is a summary of current projects. As a consequence, it contains confidential information, too. A patent application contains research results that are considered to be of high value. Therefore a patent application is top secret and accessible for only a very small fraction of the employees. In contrast to this, information in a press release is public and accessible for anyone. Finally, an internal newsletter is addressed to all employees. This newsletter contains internal information that is accessible for every employee of the company. Nevertheless, this information should only be read by employees of the company.

Level	Description
Top secret	Very sensitive information Accessible for a small fraction of the employees
Confidential	Sensitive information Accessible for some of the employees
Internal	Internal information Accessible for all employees
Public	Public information Accessible for anyone

Table 3.1: Different protection levels of documents in the scenario

Name	Abbr.	Content	Level
Project report	ProRep	Confidential research results	Confidential
Project summary	ProSu	Summary of current projects	Confidential
Patent application	PA	Top secret research results	Top secret
Press release	PreRel	Public available information	Public
Internal newsletter	IN	Information for all employees	Internal

Table 3.2: Document types used in the scenario

The subjects of our scenario act in five different roles (roles are described in [SCFY96] and explained in Section 2.1.2), namely **employee**, **researcher**, **senior researcher**, **accountant** and **senior accountant**. These roles are organized in a hierarchy as depicted in Figure 3.1. In this example, **employee** is the generic role for every employee of the company. A **researcher** is working in projects and performing research. A **senior researcher** is the supervisor of one or more researchers. In similar fashion, a **senior accountant** is the supervisor of one or more accountants, who execute administrative work.

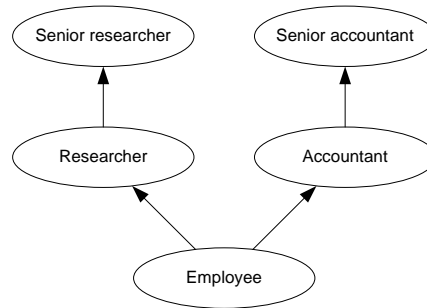


Figure 3.1: Role hierarchy of the scenario

In the following, we present different situations that illustrate challenges

for a history-based access control model. We believe that similar processes and requirements can be found in many other scenarios, e.g., in health care, public administration or academic research. The data transfers described in situations 1, 2 and 3 are illustrated in Figure 3.2.

Situation 1: *Text content of XML elements can be composed from different sources.*

Let us assume that in our scenario a project leads to some promising results, so that a press release is created (Step 1 in Figure 3.2) to announce the success to the public. The press release carefully avoids to expose any trade secrets. Shortly afterwards, a project report is written, too, consisting of several sections including an introduction and a main section. Some parts of the former press release are reused in the introduction of the project report and some parts of the introduction are also written from scratch (Step 2). As a consequence, the XML element carrying the introduction contains parts of text from different sources.

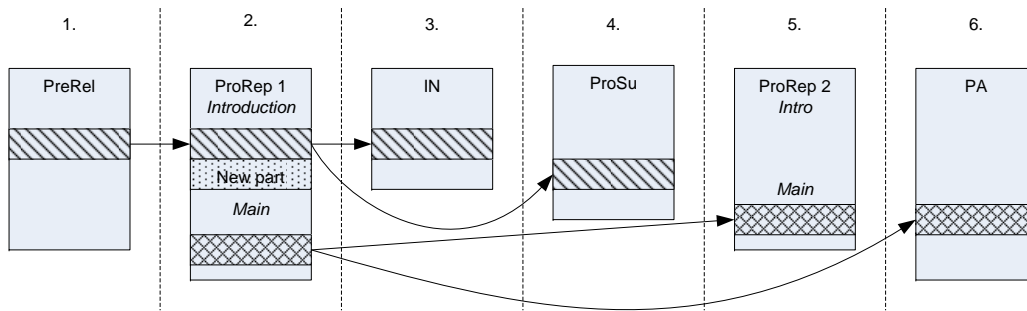


Figure 3.2: Data transfers of Situations 1, 2 and 3

Situation 2: *Allow copying depending on source of copied elements.*

Later, a company internal newsletter is published to inform all employees about the new results. For that, some parts of the project report are reused (Step 3), but only those which were extracted from the press release before, because the other parts are considered to be confidential and are therefore not allowed to be copied.

Situation 3: *Deny viewing depending on location of copied elements.*

Additionally, the same public information, which originally came from the press release, is also put in a company-wide project summary (Step 4), which contains an overview of all current projects. Other projects reuse parts of the project report as well and transfer copies of the confidential parts to the

corresponding documents (Step 5). After that, parts of the project report are copied to a patent application (Step 6). As a consequence, the protection level of these parts must be raised to top secret, until the patent application is granted. This also affects copies of that part, which reside in other project reports.

Situation 4: *Deny viewing depending on previous content of a document.*

Another situation arises when some top secret information is inserted into an existing report. At the time the information is present in the report, researchers are not allowed to view the entire report anymore. Later on, the top secret information is deleted from the report. Even though that the top secret information is now deleted, researchers are still not allowed to view the report, because they could learn from the remaining document about the very confidential information.

Situation 5: *Allow copying depending on the previous value of an attribute.*

Moreover, some projects are funded by an external company, which is denoted in the **funded-by** attribute of the corresponding reports. In our example, a policy states that results of a project funded by “Company A” are not allowed to be copied to projects funded by “Company B”. Even when A stops to fund a certain project and the attribute **funded by** now reflects a different funder, it is still prohibited to copy the corresponding content to a project report funded by B.

Situation 6: *Allow operations depending on the subject that performed a previous operation.*

Additionally, a policy states that certain modifications of a document made by a senior researcher cannot be changed by subjects in an inferior role, e.g., by a researcher. For example, a researcher can change the title of a section, e.g., to make a suggestion, until the title is changed by a senior researcher, who has the authority to declare the title as final.

Situation 7: *Allow operations depending on previous operations of the subject.*

A different situation arises, when the company of our scenario executes projects for other companies that are competitors to each other. In this case, the access control system must ensure, that a researcher only gains access to data of one of the competing companies. Otherwise a conflict of interest could occur, because a researcher could misuse the knowledge gained from the data of one company. This problem is typical for the consulting business, but in general it affects all industries of the service sector, e.g.,

the IT service industry. To enable restrictions that deny misuse of insider knowledge, the access control system must be able to grant or deny access depending on former accesses of a subject.

Situation 8: *Default policies and exceptions from policies.*

Another policy states that every employee is allowed to view company internal newsletter. Some parts of the internal newsletter can contain confidential information. Although these parts of the newsletter are not considered to be top secret, they should not be viewed by specific groups of employees. For example, some parts should not be visible for accountants, whereas other parts should not be visible for researchers. Additionally, some of these restrictions may not apply for the corresponding senior position, e.g., a senior accountant is allowed to view certain parts, which are denied for accountants. Alternatively to writing one newsletter and automatically censoring some parts of it, the company could also create several versions newsletter manually, which is clearly less efficient and less comfortable.

There are several ways to define access control rules for situations like the one sketched above. The most efficient way to specify the corresponding access control rules is to define default policies for the most common cases. In this example, all employees are allowed to view the entire newsletter. After this, each exception can be specified by an additional access control rule. For example, a rule that denies viewing of some research related information for accountants. As senior accountants are allowed to view some parts which are not allowed to be viewed by accountant, another rule can specify this exception.

The alternative to specifying both positive and negative rules, is to specify only one kind of rules, e.g., only positive permissions. The drawback of this approach is that rules get more complex. We illustrate the reasons for the increased complexity in Figure 3.3.

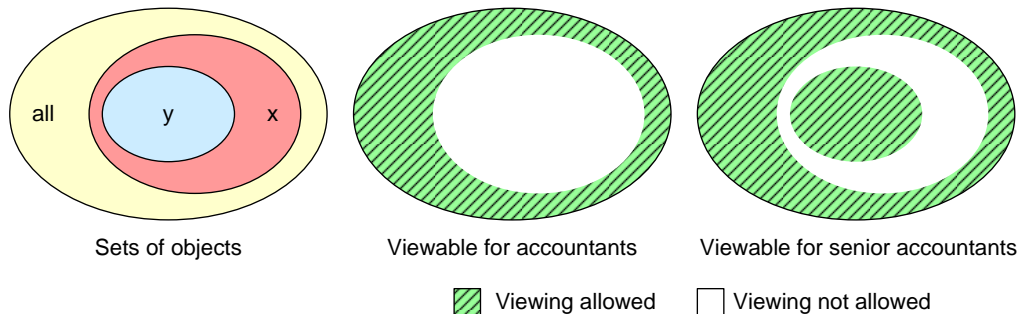


Figure 3.3: Sets of objects with different permissions

This figure illustrates three sets of objects where each set is depicted as an oval. These sets are depicted for three situations. On the left side, we show the sets in general. In all cases, the outmost oval represents all reports. The oval labeled with “x” denotes the set of objects that accountants are not allowed to view, whereas the oval labeled with “y” represents the set of objects that senior accountants are allowed to view in addition to that what accountants are allowed to view. In the middle, we show the viewable objects of an accountant and on the right we show the viewable objects of a senior accountant, where the viewable objects are indicated by the shaded area in the ovals.

If we use positive and negative rules, each rule needs to specify a set of objects corresponding to one of these ovals. In contrast to this, if we only have positive or negative rules, the specification of rules gets much more complex. For example, the rule that defines the viewable objects for accountants must specify a set of objects that corresponds to the subtraction of “x” from “all”. The rule for senior accounts is even one step more complex.

3.2 Requirements

The previous situations have illustrated several challenges for an access control system. In the following, we extract requirements from these challenges and define them individually. These situations were illustrated within our example scenario. Concerning the requirements, we abstract from our example scenario and define requirements that are useful for a wide range of scenarios that share some properties with our scenario.

Requirement 1: *Support of protection units smaller than XML elements.*

We have illustrated in Situation 1 that XML elements can contain text fragments, which differ in the way how they were created. This can be very important for security. For example, one part of an element can be created from scratch, whereas the other part is copied from a top secret document. Therefore, the access control system must keep track of these parts individually. Moreover, text fragments can differ in the subject which has created them. Consequently, we need a mechanism that keeps tracks of these parts.

Requirement 2: *Set of operations to view and edit XML documents with appropriate level of abstraction.*

We need to define a set of operations that allows to view and edit XML documents. There are many alternatives to design such a set of operations. These alternatives differ in the level of abstraction of the proposed oper-

ations. The simplest set of operations is a set consisting of a read and a write operation. The level of abstraction of this set is too low to capture the semantics of the operations in a way that is sufficient for our needs. For instance, we cannot distinguish whether elements were newly created or copied from somewhere else in case of this simple set of operations. We need to be able to differentiate whether a document part was created from scratch or whether it was copied from somewhere else. Moreover, concerning editing, we must differentiate between the creation of new data and the deletion of existing data. A simple **update** operation, which only defines that an element was changed, is not sufficient, since the aforementioned details would not be captured. Consequently, we need a **create** operation, a **delete** operation and a **copy** operation for elements and their text content. These operations must be applicable to entire XML elements or to parts of their text. We also need the **create** and **delete** operation for attributes, because we want to keep track of their creation and deletion as well. Since we regard attributes as atomic units, we model the changing of an attribute value with the operation **change-attribute**. Finally, we need a **view** operation that is used when the user wants to view a document or parts of it.

Requirement 3: *Record the source and destination of copy operations.*

We have illustrated in Situation 2 and Situation 3 that it is relevant for access control to know from where or to where certain parts of a document were copied. The source of a document part is an important aspect to determine its required level of protection. When a document is considered to be top secret, we also consider copies from it as being top secret. Moreover, to know to where a document part was copied, is important, too, especially if the destination is a confidential or top secret document. In these cases, it is desirable to restrict access to the source parts to avoid information leakage. As a result, we must record the source and destination when documents parts are copied to another location. In addition to this, we must be able to use the gathered information for access control to enable the policies mentioned in this paragraph.

Requirement 4: *Access control for the copy operation.*

In Requirement 2, we explained that we need an explicit copy operation as part of our model. To be able to define access for situations as Situation 2 and Situation 5, our model must also be able to define permissions for this operation. We formulate an individual requirement for this, since having a copy operation as part of a model does not imply that the model is able to define access for that operation. For example, the copy operation can be internally translated to a pair of read and write operations and access could

be defined only for those operations.

Requirement 5: *Recording of previous attribute values and the previous elements in a document.*

In Situation 5, we illustrated that some situations require to define access based on a previous value of an attribute. We think that many similar situations can be found in other scenarios as well. Generally speaking, the knowledge of previous attribute values helps to determine the required level of protection of a document, because they characterize a document in addition to its current content. Therefore, our model must record previous values of attributes and allow to define access based on that information.

The previous content in form of elements is of also interest, since it also helps to characterize a document. The knowledge that a specific element was contained in a document can be used to specify its required level of protection. For example, the protection level of a document can be raised if it has previously contained a top secret element.

Since we require to record the previous values of attributes and the previous elements in a document, we could also record the previous text content of these XML elements. But, here again, we assume that attributes and the text of an XML element differ in their semantics and how they are used. As stated above, we consider attribute values to have clear defined semantics and the text content to have less clear defined semantics or undefined semantics. We assume that attributes define certain properties of an element and its text content carries continuous text. Without clear defined semantics it is impractical to write access control rules. As a consequence, we do not record the previous text content of XML elements.

Requirement 6: *Record operations performed by each subject.*

We have illustrated in Situation 6 and Situation 7 that some policies depend on the operations performed by a certain subject. To enable this kind of policies, we must record the operations performed by each subject. Moreover, our model must be able to express policies based on that information.

Requirement 7: *Record the context of previous operations.*

In Situation 7 we gave an example, where access is defined depending on the subject that performed a previous operation. Besides the subject of a previous operation, other aspects can be relevant, too. We refer to these aspects as the context of an operation. This context helps to specify an object more precisely as it can make the essential difference when something was created or who has created an object. It is an open question which aspects of the context should be regarded. Our model must record the subject

that performed an operation, the role the subject was active in and the time of the operation. We believe that these aspects are the most helpful ones. Nevertheless, further aspects might be of interest to. Therefore we require that our model is extendable to handle these additional aspects, too. Our model must be able to define access depending on the recorded context information to enable policies based on that information.

Requirement 8: *Positive and negative policies.*

We have illustrated in Situation 8 that it is comfortable and efficient to have positive (allow) and negative (deny) rules. This reduces the complexity of individual rules and makes access control definition more efficient. Therefore, we require our model to support positive and negative rules.

Requirement 9: *Flexible conflict resolution strategy.*

To support the definition of default policies and corresponding exceptions, we need a flexible conflict resolution strategy that defines an order among all rules for the case where multiple rules are applicable for an object.

More specifically, the rules should be ordered in a way that a rule defining an exception has a higher priority than the corresponding default rule. In other words, the rules should be ordered depending on how specific they are. A too simple conflict resolution strategy would not allow to specify exceptions from a default rule, e.g., the strategy “deny takes precedence over allow” without further mechanisms does not allow to specify positive exceptions as we illustrated in Situation 8.

Chapter 4

Model

In this chapter, we give an overview of our model and its components, which are explained in detail in the following sections. We start with an overview of our model, continue with the operations defined in our model, go on with a description of the history and finally present the syntax for access rules. As part of the access control rules, we extend the function library of XPath. Therefore, we give details to these extension functions. We conclude this chapter, with an example of how to use our model. For this purpose, we show how we can express Chinese Wall policies with our model. In this example, we show that our model is better suited for real world scenarios than the original Chinese Wall model, since our model overcomes unnecessary restrictions of the Chinese Wall model.

4.1 Overview

Our history-based access control model defines which subjects are allowed or denied to access certain parts of an XML document. Concerning the subjects of our model, we design our model for human users. We use roles (as described in [SCFY96] and Section 2.1.2) to model these subjects. The objects of our model are different parts of an XML document. These parts can either be entire XML elements (including attributes and text content), attributes or parts of the text content of an XML element. We define a set of operations, which enables the user to view and edit XML documents. The effects of these operations are recorded in the history. When we record an operation in the history, we also log the context information of that operation. Generally speaking, context information can be any information that helps to specify the situation of the operation more precisely. In our case, we record the date and time of the operation, the subject that performed the operation

and the role the corresponding subject was active in. If it is required, this context definition can be extended. Summing up, the history stores how a document was created.

Finally, we use access control rules to define that subjects in a certain role are allowed or denied to perform a given operation on specific objects. These objects are described by a condition, which defines predicates on the content of the current document and on the history. When a user tries to perform an operation on an object, we must check whether there is an access control rule that matches with the active role of the user, the operation that he wants to perform and with the object. The first two aspects can be checked directly. In contrast to this, to check whether a rule matches with an object, we must check whether the object has the properties described by the condition of the rule. Therefore the condition must be evaluated for the current object.

4.2 Subjects

The subjects of our system are human users, which view and edit XML documents. As stated above, we use roles to model the subjects. This leads to a higher level of abstraction and therefore leads to more flexibility compared to directly listing individual subjects. There are many variations of the RBAC model, where each variation adds further enhancements to the model. Since we focus on how to specify the objects based on their history, we only use the basic version of RBAC [SCFY96]. We use the role hierarchies to solve conflicts between positive and negative rules. Nevertheless, advanced versions of the RBAC model can be combined with our model, since the modeling of subjects and objects is more or less independent of each other.

4.3 Objects

We have three types of objects in our model. The first type of model is the XML element. The second type are attributes and the third type is a text block, which we use to structure the text content of an XML element. In the following, we give details to each type of object.

4.3.1 XML elements

XML elements are one of the three types of objects in our model. The attributes and the text content of an XML element also belong to this type of object. As a consequence, if access is granted to an XML element, its

attributes and text content can also be accessed. However, in cases where finer granularity is needed, our model can address the attributes and the text content individually. The children elements of an XML element are not part of this type of object. In contrast, these are individual object themselves. Consequently, if access is granted to an element, access to its children elements is still undefined. But if access to an element is denied, access to its children elements is also denied, since XML elements require to be interpreted together with their parent element to determine the intended semantics.

4.3.2 Attributes

In cases where the granularity of elements is too coarse, our model can also address attributes as individual objects. For example, if data about persons and their personal preferences is collected to create statistics, one can grant access to a person on the level of XML elements and deny access to attributes of a person that allow to identify the person. Using this method, data can be censored to avoid loss of privacy. Moreover, by only denying access to certain attributes the structure of an XML document, which is required to interpret the elements, is maintained.

4.3.3 Text blocks

Our model splits the text of an element internally into smaller units, which we refer to as *text blocks*. These text blocks are used in this model for two purposes. First, they are one type of object, which allows to define access for individual text blocks. Second, text blocks are needed to keep track of text content, where different parts of the content were created in different ways. In other words, we also must internally use text blocks to record details of the editing process.

We need text blocks because the text content of an XML element can be composed of several parts, which were created in a different way. Their creation can differ in several aspects. Text can either be created newly or be copied from a different location. Moreover, the text can be created by different subjects or at different times. Our model internally uses text blocks to keep track of these individual parts and how they were created. Every create operation on the text of an XML element creates a new text block. In other words, when new text is added, we create a new text block. In addition to this, text blocks or parts of their text can also be deleted. If new text is added in the middle of an existing block, we split the existing block and create a new block for the added text. We use a rule, which defines the

structure of the text content in terms of text blocks. The corresponding rule is given by Definition 4.3.1. Our model enforces this rule by splitting existing text blocks and by creating new text blocks.

Definition 4.3.1 (text block). *Each part of the text content of an XML element that differs in the way how it was created or to where it was copied from the remaining text of an element, must be kept as an individual text block.*

4.4 Operations

In this section, we describe the set of operations of our model. This set of operations must have a level of abstraction that allows to keep track of the way how a document was created. We must be able to differentiate whether a document part was created from scratch or whether it was copied from somewhere else. Moreover, concerning editing, we must differentiate between the creation of new data and the deletion of existing data. Consequently, we need a **create** operation, a **delete** operation and a **copy** operation for elements and their text content. These operations must be applicable to entire XML elements or to parts of their text. We also need the **create** and **delete** operation for attributes, because we want to keep track of their creation and deletion as well. Since we regard attributes as atomic units, we model the changing of an attribute value with the operation **change-attribute**.

Finally, we need a **view** operation that is used when the user wants to view a document or parts of it. Most of the operations can be applied to elements, text and attributes. Each operation has an effect on the document itself as well as on the history. In addition to the operation itself, we also record the context of each operation. In our case, the context consists of the date and time of the operation, the subject that performed the operation and the role the corresponding subject was active in. After introducing the operations in general, we now give details on each operation.

4.4.1 Create

There are three versions of the create operation. There is one version for each type of object in our model. As stated above, these types are XML element, attribute and text. Each operation uses a specification of the destination position as first parameter. The second parameter depends on the version of the operation. All versions of the create operation and their parameters are summarized in Table 4.1.

Operation	Parameter 1	Parameter 2	Parameter 3
create element	position	element name	initial value
create attribute	position	attribute name	
create text	position	new text	

Table 4.1: Versions of the **create** operation and their parameters

The **create element** operation creates an element without any attributes or text. It needs a position, where the element should be created as first parameter and the name of the new element as second parameter.

The attributes of an element are created with the **create attribute** operation. This operation also needs a position, where the attribute should be created as first parameter and the name of the new attribute as second parameter. The initial value of an attribute is defined by the third parameter.

The **create text** operation is used to add new text to an element. This operation has an argument that specifies the position of the new text. This position can point into an existing text block, before an existing text block or after an existing block. In the first case, in which the position points into an existing text block, we split the existing block at the position where the new content should be placed and the new content is placed in-between the split blocks. This splitting is required according to Definition 4.3.1. The second argument of the **create text** operation is the new text that should be created.

4.4.2 Delete

In similar fashion to the **create** operation, there are three versions of the **delete** operation. The **delete** operation is used to delete elements, attributes, text or parts of the text. Since elements and their attributes are checked in rules, we keep them after deletion in the histories. This approach enables policies based on the former content of a document as stated as a requirement (see Requirement 5 in Section 3.2). All versions of the **delete** operation are listed together with their parameters in Table 4.2.

Operation	Parameter 1
delete element	element specification
delete attribute	attribute specification
delete text	specification of text range

Table 4.2: Versions of the **delete** operation and their parameters

The **delete element** operation deletes an element including its attributes and text content. Its only parameter is the specification of the element that should be deleted. The operation can only be applied to elements without any children. If an element has children elements, these must be deleted before the element itself can be deleted. The deleted element is stored in the history.

The **delete attribute** operation deletes an attribute. Its only parameter is the specification of the attribute that should be deleted. The deleted attribute is kept in the history.

The **delete text** operation is used to delete a range of text. In a special case, the complete text of an element can be deleted. This operation has an argument that specifies the range of text that should be deleted. Since the text of an element is structured into text blocks and we must respect Definition 4.3.1, we delete the text by deleting the corresponding text blocks. If the start or the end of the range points into a text block, we split the affected block at the corresponding position. As a result, the deletion of a range of text can be performed by deleting a range of text blocks.

4.4.3 Copy

The **copy** operation can be used for elements, text or parts of the text. We record the copying of elements and text in the history by denoting which object is a copy of which other object. All versions of the **copy** operation are listed together with their parameters in Table 4.3.

Operation	Parameter 1	Parameter 2
copy element	element specification	position
copy text	specification of text range	position

Table 4.3: Versions of the **copy** operation and their parameters

The **copy element** operation copies an existing element to a new location. The new location can be within the same document as the source element or within another document. The operation has two parameters. The first parameter specifies the existing element that should be copied and the second parameter specifies the position to where the element should be copied. The element is copied including its attributes and its text content, but without its children elements. When a complete tree of elements should be copied to a new location, the copy operation must be used for each element individually. If our model is implemented in an editing program for XML documents, the program can automatically translate the copying of a tree

into the corresponding sequence of copy operations for individual elements. The structure of the text blocks of the source element is recreated at the destination element. For each pair of text blocks, which are copies of each other, we denote that they are copies of each other in the history.

The **copy text** operation copies a range of existing text to a new location. For this purpose, it uses a specification of a text range to denote, which part of the text of an element should be copied. In a special case, the complete text of an element can be copied. Since copying text changes to where parts of the text was copied, we must split the affected blocks before we copy text to maintain Definition 4.3.1. As a consequence, we copy text by copying entire text blocks to the destination. If the start or the end of the range points into a text block, we split the affected block at the corresponding position. As a result, the copying of a range of text can be performed by copying a range of text blocks. For each pair of text blocks, which are copies of each other, we denote that they are copies of each other in the history.

4.4.4 Change Attribute

The **change attribute** operation allows users to change the value of a specific attribute. Since former values of an attribute can be checked by rules, we record the changing of an attribute in the history. The parameters of the **change-attribute** operation are shown Table 4.4.

Operation	Parameter 1	Parameter 2
change-attribute	attribute specification	new value

Table 4.4: Parameters of the **change-attribute** operation

The **change-attribute** operation has two parameters. The first parameter specifies the existing attribute that should be changed in its value. The second parameters is the new value to which the specified attribute should be set to.

4.4.5 View

The **view** operation displays elements, attributes and text content. When a user wants to view an entire document, the **view** operation must be invoked for every objects of the document. In contrast to the read operation of some other systems, e.g., [BL73, BN89], the view operation does not imply a data transfer in a technical system. In contrast to this, a view is presented to the user. Since the user could misuse the viewed information, the view operation

is logged in the history at the time when the view is created and presented to the user. All versions of the **view** operation are listed together with their parameters in Table 4.5.

Operation	Parameter 1
view element	element specification
view attribute	attribute specification
view text block	text block specification

Table 4.5: Versions of the **view** operation and their parameters

The **view element** operation is used to view an element. With this operation only the element itself can be viewed, but not its attributes and text content. This operation also does not allow to view the children elements of an element. For these elements, the **view element** operation must be invoked individually. If our model is implemented as part of an editing program, the program can invoke the **view element** operation for every element of the document the user wants to view. As a result of this process, the program, can create a view of the document.

In similar fashion, to the **view element** operation, the **view attribute** operation is used to view attributes and the **view text block** operation is used to view the text content of an XML element.

4.5 History

We use the history to keep track of changes caused by the operations **create**, **copy**, **delete**, **change attribute** and **view**. As stated above, we use the history to record how the content of a document was created. Finally, we can access this information in conditions to define the applicable objects of an access control rule.

We record the following aspects in the history:

- The context of each performed operation, where the context is a tuple of subject, role and date.
- The previous values of attributes, where each change is caused by the **change-attribute** operation.
- Deleted elements.
- Deleted attributes.

- Which elements or text blocks are copies of each other. These copies were created by the **copy** operation.

Concerning the **copy** operation, we store which object is a copy of another object. We store this information by keeping track of the *is-copy-of* relation between objects. We define the *is-copy-of* relation between two objects o_1 and o_2 as follows:

Definition 4.5.1 (*is-copy-of* relation). *An object o_1 is in is-copy-of relation with an object o_2 , if o_1 was created by applying the **copy** operation on o_2 .*

To visualize this, we can view the copied objects as a graph, where each node of the graph represents an object and each directed edge of the graph represents a copy operation. This directed edges point from the original to the copied object. In other words, each edge represents an *is-copy-of* relation between two objects of the graph.

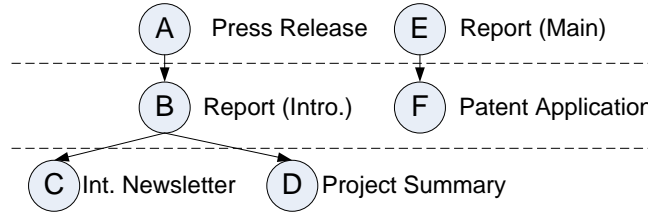


Figure 4.1: Illustration of the *is-copy-of* relation

Figure 4.1 shows a graphical illustration of the *is-copy-of* relation of the objects mentioned in the scenario. We refer to this kind of illustration as *copy graph*. We define a *copy graph* as follows:

Definition 4.5.2 (*copy graph*). *A copy graph is a graph, where each object is represented as a node and the is-copy-of relation between the objects is represented by directed edges.*

Moreover, Figure 4.1 shows two copy graphs. The copy graph on the left side of the figure shows three *is-copy-of* relations, whereas the graph on the right side of the figure only shows one such relation. In this example, all objects are text blocks. In the general case, objects of a copy graph can be XML elements as well. In addition, the copy graph in the figure may not reflect all *is-copy-of* relations and there can be nodes that are in *is-copy-of* relation with the depicted nodes, but these are not shown. In the case, where no nodes are omitted from the copy graph, we refer to the copy graph as being *complete*. Such a complete copy graph is defined as follows:

Definition 4.5.3 (complete copy graph). *A copy graph is complete, if there is no object that is in is-copy-of relation with an object of the graph and that is not represented by a node of the graph.*

4.6 Access Control Rules

In this section, we define the syntax and the semantics of access control rules. We define two types of rules. The first type of rule defines permissions for the unary operations **create**, **view**, **delete** and **change attribute**. Accordingly, we refer to this type of rule as *unary rule*. The second type of rule defines permissions for the binary operation. Consequently, we call this type of rule *copy rule*. The syntax of both types of rules is listed in Figure 4.2. We define the semantics of the rules below.

<i>Unary rule</i>		<i>Copy rule</i>	
Element	Description	Element	Description
Role	Role	Role	Role
Operation	Operation	Operation	“Copy”
Object	XPath	Object	XPath
		Destination	XPath
Mode	allow deny	Mode	allow deny

Figure 4.2: Syntax of access control rules

Access control rules consist of several fields, where each field except of the **mode** field defines a condition that must be true to apply the rule. If any of the conditions defined by a field of the rule is not satisfied, then the rule is not applicable. Next, we continue by explaining the different fields of a rule.

4.6.1 Role field

The **role** field of a rule describes the role (as described in [SCFY96] and Section 2.1.2) in which a subject must be active in to apply the rule. In other words, if a subject is active in the same role as defined in the corresponding rule, the rule is applicable for him. In addition to this, the rule is also applicable, if a subject is active in a role that is superior to the role defined in the rule. Reconsider the role hierarchy from the scenario (see Section 3.1), where **employee** is the most generic role and **researcher** is a role that is superior to **employee**. In this example, a rule defined for the role **employee** is also applicable for subjects that are active in the role **researcher**.

4.6.2 Operation field

The **operation** field of a rule defines the operation for that a rule is applicable for. In other words, the operation defined in the rule must be equal to the operation that a subject wants to perform. Otherwise the rule is not applicable for that operation. In case of a copy rule, the operation field is always **copy**. Moreover, we do not differentiate between the different types of objects in the **operation** field of a rule. For example, if the creation of an attribute should be allowed, the **operation** field of the corresponding rule is set to **create** instead of **create attribute**. To denote that the rule allows the creation of a specific attribute, an attribute must be specified in the **object** field of a rule.

4.6.3 Object field

The **object** field of a rule defines for which objects a rule is applicable for. Instead of listing individual objects in rules in an ACL-like manner [GD72], we describe objects by their properties, e.g., location within a document or attribute values. In addition to this, we can describe objects by their history. We use XPath patterns [CD99] (see also Section 2.2.5) to describe the objects for which a rule is applicable. We use XPath, since its clearly defined semantics makes the interpretation of the resulting rules unambiguous. Moreover, XPath has a predefined set of mechanisms that can be used for our purpose, which also simplifies the implementation of our model. To check whether a rule is applicable for a specific object, we must evaluate the XPath pattern for the current object. As a result, we will receive a set of objects that is specified by the XPath pattern. If the current object is part of this result set, the rule is applicable for that object.

4.6.4 Destination field

The **destination** field of a copy rule defines the destinations to where an object must be copied to be affected by the rule. Since only the **copy** operation has a destination, only copy rules have a **destination** field. When a user performs a **copy** operation, its destination must be included in the **destination** field of a copy rule to apply the corresponding rule. Similar to the **object** field, the **destination** field is an XPath pattern. This pattern must be evaluated for the current destination document, which yields to a set of objects that are described by the pattern. If the destination of the current **copy** operation is part of this result set, then the copy rule is applicable for that destination.

4.6.5 Mode field

Finally, the `mode` field of a rule does not define a condition that must be fulfilled. Instead, it defines whether the rule is positive or negative. If the `mode` field of a rule is set to “deny”, then the rule is negative. In contrast to this, if the `mode` field of a rule is set to “allow”, the rule is positive. Thus, a positive rule allows the specified operation on the object in question, whereas a negative rule denies to perform the operation.

4.6.6 Conflict resolution strategy

In some cases, there can be more than one applicable access control rule for an operation that a user wants to perform. We have no problem if all of these rules are either positive or negative. But in cases, where some of these rules are positive and some others are negative, we have a conflict. In these cases, we either must allow or deny the corresponding operation.

A common conflict resolution strategy is to define that one type of rule has priority over the other type, e.g., that deny rules always take precedence. According to Requirement 8 (see Section 3.2), we want to have a more advanced strategy, which also allows to specify positive exceptions to a negative rule. We cannot do this with the simple “deny takes precedence over allow strategy”. Therefore, we use a two step approach in conflict resolution.

In the first step, we compare the `role` fields of all matching rules. Since rules also match if the subject is in a superior role to the role defined in the rule, we can have rules which differ in their `role` field. In this case, a rule that specifies a superior role has precedence over a rule that specifies an inferior role. For example, if we have an allow rule for subjects in the role `senior accountant` and a deny rule for subjects in the role `accountant`, then the rule for subjects in the role `senior accountant` takes precedence.

We only apply the second step, if the conflict was not solved with the first step. The conflict is not solved, if there are still positive and negative rules remaining, which do not differ in their `role` field. In these cases, we finally apply “deny takes precedence over allow”.

4.6.7 Default semantics

In contrast to the previous situation, where multiple rules are applicable for an object, there also can be objects for which no access control rule defines access. In these cases, we apply the default semantics of our model, which is deny. In other words, if the access to the object is neither allowed nor denied by a rule, then the object is not accessible.

4.7 Accessing the History with XPath

We use access control rules to define access depending on the content of a document and on the history. Recall that we use XPath patterns to define the objects of a unary rule as well as the destination of a copy rule. As a consequence, we need a method to access the histories within an XPath pattern.

It is not possible to access the history information with the predefined mechanisms of XPath, because the mechanisms of XPath are restricted to the current document and at least some parts of the history should not be stored within the document itself, e.g., the information about the is-copy-of relation among elements should be stored separately, since it affects multiple documents. Moreover, the access to the histories should be independent of the method of storing the histories. This allows to change the storing mechanism without the need to change all existing access control rules. As a consequence, to enable access to the history information from XPath patterns, we extend the function library of XPath with a set of functions that we define in the following sections. The functions are designed to enable access to all history information (see Section 4.5) that we record. Since XPath allows to combine several aspects in a logical condition, we can also define rules which need to use multiple aspects of the history. All functions can optionally use an element as first parameter. Otherwise, all functions apply to the current context element. We have organized our functions in six different groups, where each group consists of functions for a similar purpose.

4.7.1 Getting Copies of an Object

This group of functions is related to the is-copy-of relation of objects among each other, where an object can either be a text block (see Section 4.3) or an XML element. The functions of this group are required to express rules that define access depending on the source of an object or on the locations to where an object was copied. Figure 4.1 illustrates the is-copy-of relation, where the examples show the processes of the scenario in Chapter 3. For each object, we denote in which document it is contained, e.g., object A is created within a press release.

The function `copies` returns all elements of the corresponding complete copy graph, whereas the function `predecessors` returns all elements that are on the path to the root element of the copy graph starting from the current element. Finally, the function `successors` returns all elements in the subtree below the current element. The elements occurring as the result of any such function are sorted ascending by creation time, which enables the author of a

rule to use the indexing mechanisms of XPath to address a specific element, e.g., `predecessors()[1]` refers to the root of a copy graph.

Moreover, the surrounding XML document of the returned elements can be analyzed in a rule. For example, to check in which type of document the copies of an element reside, one must retrieve the corresponding root element of a copied element. As a result, the access rights of one XML document can depend on the content of several other XML documents. We refer to these documents as *dependent documents*. We define whether one document depends on another document as follows:

Definition 4.7.1 (depending document). *A document d_1 depends on the document d_2 if at least one object of d_1 is part of the complete copy-graph of an object in d_2 .*

Note that the relation defined above is symmetrical, which means that if d_1 depends on d_2 then d_2 also depends on d_1 . For one document d_1 there can be more than one document on which d_1 depends on. We refer to all documents on which the access rights of one document can depend as the set of depending document, which we define as follows:

Definition 4.7.2 (depending documents). *The set of depending documents of a document d_1 , is the set of all documents d_x , where d_1 depends on d_x .*

The depending document are important, when the access rights of one document are evaluated, since the content of the depending documents can be relevant for the access rights of the document in question.

Table 4.6 lists the functions for getting the copies of an object and gives an example for each one. These examples refer to the left graph of Figure 4.1 and B as the current element.

Function	Returns	Example
<code>copies()</code>	all elements of the copy graph	{A, B, C, D}
<code>predecessors()</code>	elements on the path to the root	{A}
<code>successors()</code>	elements in the subtree below	{C, D}

Table 4.6: Getting the copies of an object

We can formulate a policy from our scenario in Chapter 3 by writing a corresponding access control rule with the help of the `copies` function. The rule denies the viewing of reports for a subjects in the role “researcher” if they contain nodes that have been copied to or from a patent application (PA). To keep the example rule short, we do not check whether the PA is still

Role:	researcher	Operation:	view
Object:	/Report//*[count(copies() [/PA]) > 0]		
Mode:	deny		

Figure 4.3: Rule denying view

pending. This would require another term in the condition which inspects the value of the corresponding attribute of the PA.

The corresponding rule is depicted in Figure 4.3. The object field uses an XPath pattern, where the path **/Report/*** denotes that this rule is applicable to any node of any report. In other words, we do not restrict objects to be located at a specific position within a report. The following square brackets define a condition, which uses the **count** function (predefined in XPath) to count the number of copies of the current object that reside in a patent application. As an argument for the **count** function, we use the **copies** function to retrieve all copies of that element in other documents. The result of the **copies** function is filtered by another condition in square brackets. This condition retrieves the root element of the corresponding document and checks whether its element name is **PA**. Finally, we check whether the result of the **count** function is greater than 0. In this case, the object matches with the XPath pattern and the viewing of this object is denied for any subject in the role **researcher**.

4.7.2 Getting Related Nodes Depending on Time

This group of functions retrieves nodes addressed relatively to the context node that existed within a specified time interval. We need this group of functions, since we store deleted nodes in the history and we want to be able to access these deleted nodes in conditions. XPath offers functions to retrieve nodes addressed relatively to the context node, but without the specification of a time interval within which the nodes have existed, since XPath only considers the current state of a document. This time interval is required to select related nodes depending on time when they have existed. Therefore, each of these functions can have a time interval as parameter, e.g., **childrenAt(t1, t2)** returns all nodes that were children of the context node in the time interval between **t1** and **t2**. To inspect a single point in time, **t2** can be omitted. If no parameter is specified, the entire lifetime of the corresponding document is inspected. When the entire lifetime of a document is inspected, all current elements, as well as all deleted elements are returned. As mentioned, all functions can optionally have an element as first parameter. The functions for getting related nodes depending on time

are listed in Table 4.7.

parentAt	followingAt	precedingSiblingAt
rootAt	precedingAt	followingSiblingAt
childrenAt	descendantAt	selfAt

Table 4.7: Getting related nodes depending on time

We illustrate the usage of the functions defined above by an example rule, which is depicted in Figure 4.4. Recall the policy from our scenario that denies the viewing of a report if it has contained an element from a patent application. To do this, the XPath pattern of the corresponding rule inspects all descendants of the root element of the report in question by using the `descendantAt` function. By definition, this function returns all descendants including deleted nodes. These nodes are filtered by a condition, which inspects the elements that are in is-copy-of relation with the element in question. We count whether at least one of these elements is contained in a patent application (abbreviated by PA).

Role:	researcher	Operation:	view
Object:	/Report/descendantAt() [count(copies() [/PA]) > 0]		

Figure 4.4: Rule denying view

4.7.3 Getting the Context of a History Entry

This group of functions offers access to the context of a specific history entry. Each function except of `getAttrChangeContexts()` returns an element consisting of subject, role and time. In addition to this, the function `getAttrChangeContexts()` also delivers the value to that the corresponding attribute was changed to. Moreover, we offer functions to retrieve the context of the creation, the deletion and the viewing of nodes. Table 4.8 lists the functions for retrieving the context of a history entry.

We illustrate the usage of the functions defined above by an example rule. Recall the policy from the scenario stating that a researcher cannot delete a section that was created by a senior researcher. The corresponding rule is depicted in Figure 4.5 and uses the `getCreationContext` function to check whether the role of the subject that created the **Section** is **senior researcher**. In that case, it denies the deletion of **Section** elements for subjects in the role **researcher**.

Function	Returns context(s) of
<code>getCreationContext()</code>	creation of current element
<code>getDeletionContext()</code>	element deletion
<code>getViewsContexts()</code>	viewing a node
<code>getAttrChangeContexts()</code>	changing an attribute

Table 4.8: Getting the context of a history entry

Role:	<code>researcher</code>	Operation:	<code>delete</code>
Object:	<code>//Section[getCreationContext() /role = 'senior researcher']/*</code>		
Mode:	<code>deny</code>		

Figure 4.5: Rule denying deleting of a section

4.7.4 Getting Accessed Nodes

This group of functions is used to get all nodes which have been accessed by a specified user or by a user in a certain role. For example, these functions are required to express Chinese Wall policies [BN89]. The functions are **created**, **viewed**, **accessed**, **deleted**, **changedAttr** and **copied**. Each function refers to a specific operation, e.g., **viewed** returns viewed nodes. In addition, the function **accessed** returns all accessed nodes independently of the operation and **modified** returns nodes that have been modified. All functions have two parameters that define conditions on the returned nodes. The first parameter **user** specifies to return only nodes that have been accessed by the specified user. Analogously, we define the parameter **role**. Both parameters can be set to **any** to indicate to return nodes accessed by any user or in any role. Optionally, each parameter can be set to **current**. In this case, the current user or his current role is used for the check. For example, **created(any, current)** returns all nodes which have been created by users who were active in the same role as the one in which the current user is active in. The functions of this group are summarized in Table 4.9. We present example rules using the functions of this group in Section 4.8.

4.7.5 Getting Specific Nodes of Current Rule

We define three functions for accessing specific nodes within an XPath pattern. The function **currentNode** returns the node in question for which the XPath pattern is evaluated. This function is required when the pattern's context changes to a document that is different from the document for which

Function	Returns
<code>created()</code>	all created nodes
<code>viewed()</code>	all viewed nodes
<code>accessed()</code>	all accessed nodes
<code>deleted()</code>	all deleted nodes
<code>changedAttr()</code>	all changed attributes
<code>copied()</code>	all copied nodes

Table 4.9: Getting accessed nodes

the pattern was initiated. The function `srcNode` retrieves the source node in question when checking a copy rule. In a similar fashion, the function `destNode` returns the destination node of a copy rule. The last two functions are necessary to define copy rules which compare the source and destination objects with each other. The functions of this group are listed in Table 4.10. We present example rules using the functions of this group in Section 4.8.

Function	Returns
<code>currentNode()</code>	inspected node of a unary rule
<code>srcNode()</code>	inspected source node of a copy rule
<code>destNode()</code>	inspected destination node of a copy rule

Table 4.10: Getting specific nodes of a rule

4.7.6 Additional Extension Functions

This group of functions are required for rules that need to inspect the current subject, the role of the current subject or whether a specific node is deleted. The functions of this group are `currentSubject`, `currentRole` and `isDeleted`. These functions provide information that is not available within XPath patterns and can be required in a variety of rules. The functions of this group are summarized in Table 4.6.

4.8 Modeling Chinese Wall policies

We conclude this chapter by presenting three example rules that demonstrate how our model can be used to express the policies of the Chinese Wall model mentioned in the scenario in an effective and flexible way. We show that

Function	Returns
<code>currentSubject()</code>	the current subject
<code>currentRole()</code>	the role of the current subject
<code>isDeleted()</code>	true if the node is deleted, otherwise false

Figure 4.6: Additional functions

our model is better suited for real world scenarios, since it can avoid the unnecessary restrictions of the original Chinese Wall model.

The Chinese Wall model (described in detail in Section 2.1.3) includes two policies to define access, namely the “read policy” and the “write policy”. The first policy is used for read and execute access and states that access to an object is only granted if the accessing subject did not access any object of the same conflict-of-interest class that belongs to a different company. As a consequence, after an access to one object, all objects of the same class that belong to different company data sets are inaccessible for that subject.

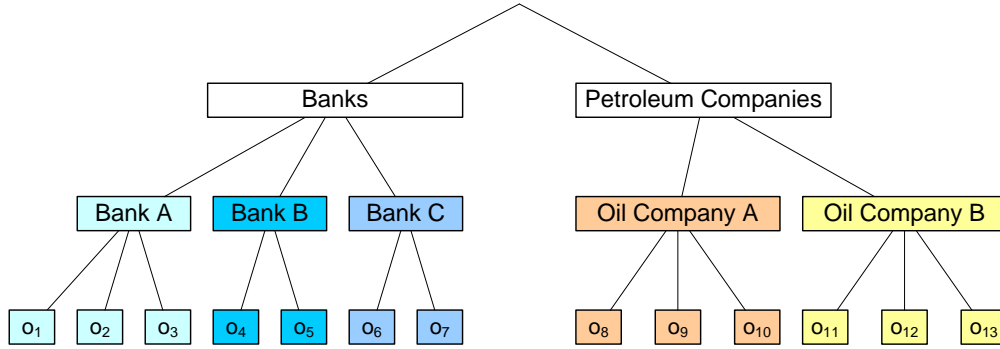


Figure 4.7: An example of the objects in the CWM

The read policy alone is insufficient, because unwanted data transfers that enable misuse of insider knowledge are still possible via multiple steps. Consider the following example depicted in Figure 4.7. We have two conflict-of-interest classes, namely banks and oil companies. A subject s_1 reads object o_1 , which belongs to “Bank A” and writes the corresponding data to object o_8 , which belongs to “Oil Company A”. After that, the subject s_2 reads that data from o_8 and writes it to object o_4 , which belongs to “Bank B”. This transfer is unwanted since o_4 belongs to a competitor of “Bank A”. As a consequence, the “write policy” is defined to prevent these unwanted transfers. It allows to write data to an object only if the previous read accesses refer to the same company data set or the read information is public. Next, we model the policies of the Chinese Wall model with our history-based approach.

We assign a conflict-of-interest class and a company to each document by adding the attributes `class` and `company` to its root element. This approach is reasonable, since each document typically refers to one company. Assigning these attributes to each element can lead to ambiguous semantics regarding the part-of relation of elements which are assigned to different companies. For example, if a “Bank A” element has children that are assigned to “Oil Company B” it is unclear whether these children are regarded to belong to “Bank A” because they are part of a “Bank A” element.

Next, we define a set of rules which allow users to execute all operations on any object, e.g., `{employee, view, allow, /*}`. Then, we specify two rules that take precedence over the allow rules and deny accesses that are not permitted according the policies of the Chinese Wall model. The first rule corresponds to the read policy and is depicted in Figure 4.8. This rule denies view access to non-public documents of different companies belonging to the same conflict class.

Role: <code>employee</code>	Operation: <code>view</code>
Object: <code>/*[count((viewed(current, any)[(//*@company != root(current-node())/@company) and (//*@class = root(current-node())/@class) and (//*@class != 'public')]) > 0]</code>	
Mode: <code>deny</code>	

Figure 4.8: Rule enforcing the read policy

The XPath expression of this rule first matches with any object (`/*`) and then specifies a condition on them. This condition states that the number of elements viewed by the current subject matching a second condition must be greater than zero. The second condition checks three properties of each viewed object. First, its company must be different from the company of the current node. Second, its class must be the same as the class of the current node and third its class must not be public. The rule matches if all these properties are fulfilled. The attributes `company` and `class` are accessed in the expression using built-in functions of XPath, functions defined by us and the child axis of XPath, e.g., `root(current-node())/@company` accesses the company attribute of the currently inspected node. Next, we present the rule enforcing the write policy in Figure 4.9.

The rule above reflects the write policy as close as possible. The write operation of the Chinese Wall model can be used to create new data or to transfer data. Since the write policy aims at denying transfers of data, we

Role: <code>employee</code>	Operation: <code>copy</code>	Source: <code>/**</code>
Destination: <code>/**[count((viewed(current, any)[(*)/@class != root(dest-node())/@class) and (*)/@class != 'public']]) > 0]</code>		
Mode: <code>deny</code>		

Figure 4.9: Strict rule enforcing the write policy

have defined a copy rule for that purpose. This allows users to modify documents belonging to different classes, which is not possible in the Chinese Wall model, because the denial of write operations also prevents the creation and modification of data. However, the rule denies transfers of public information that resides in a non-public document, although the transfer of public information should not be restricted.

Moreover, transfers to documents of different conflict classes are denied as well to prevent transfers via multiple steps, although the resulting state of only one such transfer is not unwanted, e.g., the transfer from “Bank A” to “Oil Company A” is acceptable if the transferred data is not transferred by another subject to “Bank B” afterwards.

To overcome these shortcomings, we present a more flexible version of the second rule in Figure 4.10. With the help of history information, we keep track of the source of information and deny unwanted transfers via multiple steps.

Role: <code>employee</code>	Operation: <code>copy</code>	Source: <code>/**</code>
Destination: <code>/**[(*/@company != root(copies(src-node())[0])/@company) and (*)/@class = root(copies(src-node())[0])/@class) and (root(copies(src-node())[0])/@class != 'public')]</code>		
Mode: <code>deny</code>		

Figure 4.10: More flexible rule for the write policy

The difference between the rule in Figure 4.10 and the rule in Figure 4.9 is that the latter rule inspects the document from where the object to be transferred came from. This rule uses the expression `copies(src-node())[0]` to retrieve the original node of which the current source node is a copy. This leads to different semantics, where the source document is relevant for the attributes `class` and `company` of an object rather than the document where the node is currently located. As a consequence, public information can be

transferred from any document to another, even if it is currently part of a non-public document. Moreover, document parts can be exchanged between documents, which are not in conflict with each other while unwanted transfers are still denied.

The example above demonstrates that our model can express history-based policies in a flexible way. Our model provides the same level of security as the original Chinese Wall model while being less restrictive. Thus, we believe that our model provides a flexible and expressive method to define access depending on histories.

4.9 Summary

In this chapter, we introduced our model for history-based access control. The objects of our model are either XML elements, attributes or so-called text blocks. Text blocks are parts of the text content of an XML element, which differ in their way of creation or to where they were copied. We define that our model automatically keeps track of these text blocks.

We described the set of operations of our model, which are **create**, **delete**, **view**, **copy** and **change-attribute**. All of these operations except from the last one can be applied to all three types of objects, whereas the **change-attribute** operation can only be applied to attributes.

The operations of our model have a high level of abstraction, which allows to keep track of many important details of the editing process. We record the details of the editing process in the so-called “History”. In the history, we record the context of each performed operation, where the context is a tuple of subject, role and date, the previous values of attributes, deleted elements, deleted attributes, and which elements or text blocks are copies of each other. These copies were created by the **copy** operation.

To record the last aspect, we defined the *is-copy-of* relation between two objects. Two objects are in *is-copy-of* relation with each other, if one object was created by applying the **copy** operation on the other. To visualize this, we introduced the copy graph, where each node of the graph represents an object and each directed edge of the graph represents a **copy** operation. If no *is-copy-of* relation is missing in the graph we call it a complete copy graph.

We use access control rules to define access. For this purpose, we presented two types of rules, namely unary rules and copy rules. The last type defines access for the **copy** operation, whereas the first type defines access for all remaining operations. We model the subjects in rules using the role-based approach. In addition to this, each rule defines access for a specified operation only. We use XPath patterns to describe the objects for which a

rule is applicable. These patterns define objects by their properties, which includes the content of objects as well as their history. For copy rules, we also use an XPath pattern to describe the destination. Finally, a rule can be either positive (allow) or negative (deny), which is specified by the mode of the rule. If conflicts between positive and negative rules occur, we take the rule with the more special role and finally apply deny takes precedence over allow. If no rule matches for an object, access is denied.

We extended the function library of XPath to access history information in XPath patterns. For this purpose, we have defined functions in six different groups. We have functions that retrieve the copies of an object, functions that can inspect deleted nodes, functions that return the nodes, which a specified subject has accessed, functions that return the context of a previous operation, functions to get specific nodes of the current rule and some additional helper functions. All these functions allow us to define objects depending on their history.

Finally, we presented an example, which shows how our model can be used to model Chinese-Wall policies. In this example, we showed that our model is suited better for real world scenarios, since it avoids the unnecessary restrictions of the original Chinese-Wall model.

Chapter 5

System Architecture

In this chapter, we present a system architecture for applying history-based access control in a distributed environment where multiple users can edit documents concurrently. The components of the system architecture are explained in detail in the following sections. Additionally, we describe the algorithms and protocols that are required for the interaction between the components. Before we start with the description of our system architecture, we will explain the challenges concerning its design.

Applying our model in a scenario where multiple users concurrently edit multiple documents introduces four challenges that are caused by our history-based access control model. There can be additional challenges, but we will only discuss the challenges that are introduced by our model and its way of defining access rights.

First, access rights of one document can depend on the content of other documents, which we refer to as depending documents (see also Section 4.7.1). Since, we assume that documents are edited in a distributed fashion, we need a method for accessing these distributed documents when calculating access rights.

Second, changes to one document require the recalculation of the views of all dependent documents, which are currently viewed. The straight forward approach for this is to recalculate the views of all dependent documents after a document has been changed. However, this results in a much higher number of view recalculations compared to models which only define access depending on the currently edited document. For example, editing 20 depending documents concurrently, leads to a 20 times higher number of view recalculations with the straight forward approach. Therefore, we need a method which reduces the number of these view recalculations.

Third, the changes of one user to a document can revoke the access rights of other users which are currently editing dependent documents. As a con-

sequence, access rights can be revoked during an editing process, which can lead to conflicts regarding the content of the document and the access rights. Consequently, we need a method for handling these conflicts.

Fourth, the aforementioned straight forward approach causes intermediate editing steps to become relevant for access decisions of other users, which is not desired. For example, a user can change a policy relevant element of a document by first deleting it and then replacing it with an updated version afterwards. In this example, the first step can revoke the access rights of another user, whereas the second step might restore these access rights. As a consequence, we need an approach that avoids this problem, more precisely, we want to avoid that intermediate editing steps become policy relevant for other documents.

In the following sections, we present a system architecture that solves the presented challenges. We start with an overview of this architecture. For the moment, we make no assumptions how the components are distributed of different physical machines. In Section 5.3, we discuss different ways of distributing the components of our architecture on different physical machines and finally present a distributed system architecture.

5.1 Architecture Overview

Our system architecture and its components are depicted in Figure 5.1. Our system uses four databases. The document database (Doc DB) contains all documents of the system. The rule database (Rule DB) contains the access control rules, which specify allowed or denied accesses to the documents and their parts. The copy database (Copy DB) stores the is-copy-of relation of the objects. Finally, the user database (User DB) stores the credentials of the users of the system as well as the corresponding roles including their hierarchy.

The user interface (UI) presents documents to the user and offers operations that can be performed on the documents. If the user invokes such an operation, the corresponding request is sent to the document processor (DP), which performs the requested operation if it is permitted. Inside the DP, the policy enforcement point (PEP) intercepts each operation and asks the policy decision point (PDP) whether the requested operation is allowed. The PDP uses the four databases to decide whether to allow or deny the requested operation. In the following, we explain the workflow for editing a document to illustrate the processes within our architecture.

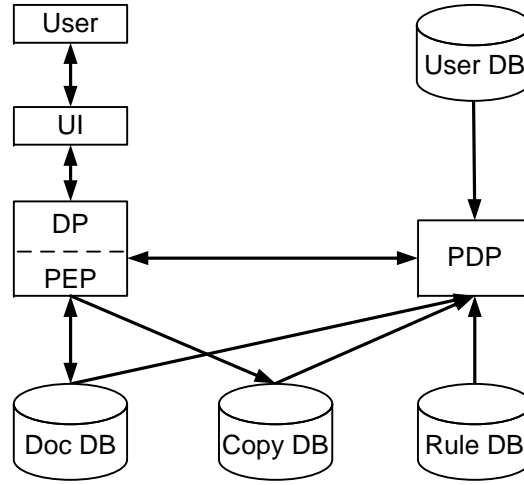


Figure 5.1: System architecture

5.2 Workflow

A document must be opened before it can be viewed or edited. Therefore, the UI offers a command to open a document. This command is sent to the DP, which loads a copy of the document from the document database. We refer to this process as *check-out*, since it has semantics similar to the check-out command of a version control system [Tic85]. After the check-out, the user can edit the document by applying the operations of our model. The changed content of an opened document including the corresponding history becomes relevant for access decisions of other documents after it is checked-in. Up to then, the content of the opened document is only relevant for access decisions concerning that document itself. The document and the corresponding history are kept as a local copy in the DP. To *check-in* a document, the user must invoke the corresponding command of the UI. Then, the DP stores the copy of the document back to the document database.

The check-in and check-out concept is more efficient and offers a higher usability compared to directly working on the policy-relevant version of a document. The first concept is more efficient, because changed content must be propagated less often, more precisely, only when a document is checked-in compared with immediately after each change. This also reduces the overhead for recalculating permissions. The usability is also higher, because of the transaction semantics of the approach. With this concept a user can decide when the changing of a document is done, instead of having potentially unwanted intermediate states to get relevant for access decisions. With this concept we give a solution for the second and fourth challenge mentioned in

the introduction of this chapter.

5.2.1 Check-out

When the user $user_1$ invokes the command to check out the document doc_a , the DP first loads a copy of doc_a from the Doc DB. The Doc DB maintains a list $list_i$ for each document doc_i that denotes by which users doc_i is currently opened to support concurrent access to documents. For that purpose, the DP adds $user_1$ to $list_a$. Next, the DP sends a copy of doc_a and the credentials of $user_1$ to the PDP to retrieve a view of doc_a for $user_1$. To create this view, the PDP performs Algorithm 1.

This algorithm removes nodes from the document for which the user in question has no view permission. For that purpose, the algorithm adds a marker to each node which is set initially to “default”, where a node can either be an element, an attribute or a text block. In line 3, we sort all rules by their role, where superior roles are placed before inferior roles. If the role field of rules is identical or incomparable, we place deny rules before allow rules. This sorting implements the conflict resolution strategy described in Section 4.6.6. The loop in lines 4 to 12 iterates over all existing rules. The condition in line 5 skips rules that are not applicable. In line 7, the XPath expression for the object of the current rule is evaluated. The result of this step is a set of nodes that match with the current XPath expression, which defines the applicable objects of the rule. For each of these nodes, the marker is set according to the mode field of the current rule (line 8 to 10). In case all markers of the document are set to a value different from “default”, we can stop evaluating rules and exit the loop (lines 11 and 12). Finally, we remove every node with a marker set to “default” or “deny” (lines 13 to 15). After that, the PDP sends the view to the DP, which forwards it to the UI (line 16).

5.2.2 Editing

To edit a document, the user first selects an operation offered by the UI, which is sent to the DP, where the PEP intercepts the operation to check whether it is allowed. For that purpose, the PEP sends the requested operation together with the current document to the PDP, which evaluates the rules to answer the request of the PEP. The current document must be sent to the PDP, since the current content of the document is relevant for its own access rights. Algorithm 2 performs the evaluation of the rules for a specific operation.

Algorithm 1: Create View

Input : $\text{rules}_{\text{all}}$, $\text{role}_{\text{curr}}$, role_hierarchy , doc
Output: doc

- 1 add marker to every node of doc
- 2 set marker of every node of doc to “default”
- 3 sort $\text{rules}_{\text{all}}$ by role (special first) and mode (deny first)
- 4 **for** *each* rule_i of $\text{rules}_{\text{all}}$ **do**
- 5 **if** *operation of* rule_i *is not “view” or role of* rule_i *is not inferior or equal to* $\text{role}_{\text{curr}}$ **then**
- 6 └ continue with next iteration of loop
- 7 $\text{nodes}_{\text{result}} \leftarrow$ evaluate XPath of rule_i for doc
- 8 **for** *each* node_j of $\text{nodes}_{\text{result}}$ **do**
- 9 **if** *marker of* node_j *is “default” then*
- 10 └ set marker of node_j to mode of rule_i
- 11 **if** *all markers of* doc *are different from “default” then*
- 12 └ exit loop
- 13 **for** *each* node_j of doc **do**
- 14 **if** *marker of* node_j *is “default” or “deny” then*
- 15 └ remove node_j and subtree below from doc
- 16 **return** doc

The algorithm for rule evaluation sorts all rules like the previous algorithm (line 1). Then, it checks the applicability of each rule by inspecting its role and its operation (line 3). For each rule, the XPath pattern is evaluated (line 5) to check whether it matches with the object in question. In case of a copy operation, the XPath pattern for the destination is evaluated (lines 8 to 11), too. If the rule is applicable, its mode is returned. After evaluating all rules, the algorithm returns “deny”, if none of the rules was applicable. The PDP sends the result of this algorithm back to the DP. If the result is deny, the DP does not perform the requested operation and informs the user via the UI. If the result is allow, the DP performs the requested operation. For that purpose, it executes the algorithm for the selected operation. We discuss these algorithms in the following.

Algorithm 2: Evaluate Rules

Input : $\text{rules}_{\text{all}}$, $\text{role}_{\text{curr}}$, role_hierarchy , op , doc , obj , doc_{dest} , obj_{dest}
Output: deny | allow

- 1 sort $\text{rules}_{\text{all}}$ by role (special first) and mode (deny first)
- 2 **for** each rule_i of $\text{rules}_{\text{all}}$ **do**
- 3 **if** operation of rule_i is not op **or** role of rule_i is not inferior or equal to $\text{role}_{\text{curr}}$ **then**
- 4 continue with next iteration of loop
- 5 $\text{nodes}_{\text{result}} \leftarrow$ evaluate XPath for object of rule_i for doc
- 6 **if** obj is not contained in $\text{nodes}_{\text{result}}$ **then**
- 7 continue with next iteration of loop
- 8 **if** op is “copy” **then**
- 9 $\text{nodes}_{\text{result}} \leftarrow$ evaluate XPath for destination of rule_i for doc_{dest}
- 10 **if** obj_{dest} is not contained in $\text{nodes}_{\text{result}}$ **then**
- 11 continue with next iteration of loop
- 12 **return** mode of rule_i
- 13 **return** “deny”

Since performing an operation can lead to modifications of view permissions, the DP asks the PDP to update the view as described above. The updated view is presented to the user via the UI.

Create

The create operation can be used to create elements, attributes or text. In all cases, we add history information that describes the operation. Algorithm 3

depicts the creation of a new element, where elem_{dst} represents the position of the new element. After creating the element, the corresponding history entry is created (line 3) and added to the doc_{dst} . Since the algorithm for creating a new attribute is very similar the one for creating a new element, we refrain from presenting it.

Algorithm 3: Create Element

Input : doc_{dst} , elem_{dst} , subj, role, $\text{time}_{\text{curr}}$

Output:

- 1 $\text{elem}_{\text{new}} \leftarrow$ create new element at elem_{dst} in doc_{dst}
 - 2 create element history for elem_{new}
 - 3 **create_hist_entry**(“Create elem.”, subj, role, $\text{time}_{\text{curr}}$)
-

The DP performs Algorithm 4 to add new text to the element elem_{dst} of the document doc_{dst} at the position $\text{pos}_{\text{insert}}$. First, Algorithm 5 is invoked to split elem_{dst} at $\text{pos}_{\text{insert}}$. After that, a new text block is created to which the new content is added (line 2). In line 3, the corresponding history entry is created.

Algorithm 4: Create Text Content

Input : doc_{dst} , elem_{dst} , subj, role, $\text{time}_{\text{curr}}$, content, $\text{pos}_{\text{insert}}$

Output:

- 1 **split_block**(doc_{dst} , elem_{dst} , $\text{pos}_{\text{insert}}$)
 - 2 $\text{block}_{\text{dst}} \leftarrow$ **create_block**(doc_{dst} , elem_{dst} , $\text{pos}_{\text{insert}}$, content)
 - 3 **create_hist_entry**(“Create text”, $\text{block}_{\text{dst}}$, subj, role, $\text{time}_{\text{curr}}$)
-

Algorithm 5 performs the splitting of text blocks, which is needed when text is created, transferred or deleted.

If the position $\text{pos}_{\text{split}}$ at which the new text $\text{block}_{\text{old}}$ should be added is not within an existing text block, then the algorithm returns immediately. In the other case, the existing text block $\text{block}_{\text{old}}$ has to be split at $\text{pos}_{\text{split}}$. Therefore, we remove the content after $\text{pos}_{\text{split}}$ from $\text{block}_{\text{old}}$ (line 5). This removed content is added to the new text block $\text{block}_{\text{new}}$ which is positioned after $\text{block}_{\text{old}}$ (lines 6 and 7). Next, $\text{block}_{\text{new}}$ gets a copy of the history of $\text{block}_{\text{old}}$ (lines 9 and 10). If the split block was in a is-copy-of relation with other blocks, the effect of the splitting has to be captured. For this purpose, the DP copies all edges which either point to $\text{block}_{\text{old}}$ (line 11) or originate from it (line 13) and modifies them to point (line 12) or originate from $\text{block}_{\text{new}}$ (line 14). The DP stores these edges in its internal memory until the document is checked-in.

Algorithm 5: Split Text Block**Input** : $\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{split}}$ **Output**:

```

1 if  $\text{pos}_{\text{split}}$  does not point within a block then
2   return
3  $\text{block}_{\text{old}} \leftarrow$  text block where  $(\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{split}})$  point into
4  $\text{content} \leftarrow$  text content of  $\text{block}_{\text{old}}$  from  $\text{pos}_{\text{split}}$  to end
5 delete text content of  $\text{block}_{\text{old}}$  from  $\text{pos}_{\text{split}}$  to end
6  $\text{pos}_{\text{new}} \leftarrow$  end of  $\text{block}_{\text{old}}$ 
7  $\text{block}_{\text{new}} \leftarrow \text{create\_block}(\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{new}}, \text{content})$ 
8 get subj, role, time from history entry of  $\text{block}_{\text{old}}$ 
9  $\text{entries} \leftarrow$  copy of history entries for  $\text{block}_{\text{old}}$ 
10 modify entries to reference  $\text{block}_{\text{new}}$ 
11  $\text{edges}_{\text{to}} \leftarrow$  copy of edges of Copy DB that point to  $\text{block}_{\text{old}}$ 
12 modify all edges in  $\text{edges}_{\text{to}}$  to point to  $\text{block}_{\text{new}}$ 
13  $\text{edges}_{\text{from}} \leftarrow$  copy of edges of Copy DB that originate from  $\text{block}_{\text{old}}$ 
14 modify all edges in  $\text{edges}_{\text{from}}$  to originate from  $\text{block}_{\text{new}}$ 
15 add  $\text{edges}_{\text{to}}, \text{edges}_{\text{from}}$  to global set of temp. edges

```

Copy

The copy operation can be used to copy elements or text. The first task is carried out by Algorithm 6. We first create a new element at the destination document doc_{dest} by applying Algorithm 3 (line 1). After that, the DP captures this transfer by adding a corresponding edge into its internal memory (lines 2 and 3). The transferred element elem_{dst} gets all attributes including their values of its source element elem_{src} (lines 4 to 6).

Algorithm 7 performs the copying of text. Since this operation has to be captured, the DP splits both the text blocks at the source document (lines 1 and 2) and the text blocks at the destination document (line 3). In line 6, we create a new block ($\text{block}_{\text{dst}}$) at the destination document (doc_{dest}) for each of the transferred blocks ($\text{block}_{\text{src}}$). We create and store the corresponding edge reflecting the transfer of $\text{block}_{\text{dst}}$. The new block gets its first history entry describing its creation context in line 9.

Change Attribute and Delete

Changing an attribute value is also logged with an entry in the history. The corresponding algorithm is similar to Algorithm 3. The delete operation can be applied to elements, attributes and text. Since the required algorithms

Algorithm 6: Copy Element

Input : $\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}$ **Output:**

- 1 **create_element**($\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}$)
 - 2 $\text{edge}_{\text{new}} \leftarrow \text{create_edge}(\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{doc}_{\text{dst}}, \text{elem}_{\text{dst}})$
 - 3 add edge_{new} to global set of temp. edges
 - 4 **for each** attr_i **of** elem_{src} **do**
 - 5 $\text{create_attribute}(\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}, \text{name of attr}_i)$
 - 6 $\text{change_attribute}(\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}, \text{name of attr}_i, \text{value of attr}_i)$
-

Algorithm 7: Copy Text Content

Input : $\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{text}_{\text{start}}, \text{text}_{\text{end}}, \text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{insert}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}$ **Output:**

- 1 **split_block**($\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{text}_{\text{start}}$)
 - 2 **split_block**($\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{text}_{\text{end}}$)
 - 3 **split_block**($\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{insert}}$)
 - 4 **for each** $\text{block}_{\text{src}}$ **within** $\text{text}_{\text{start}}$ **and** text_{end} **of** elem_{src} **do**
 - 5 $\text{content} \leftarrow \text{text content of block}_{\text{src}}$
 - 6 $\text{block}_{\text{dst}} \leftarrow \text{create_block}(\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{pos}_{\text{insert}}, \text{content})$
 - 7 $\text{edge}_{\text{new}} \leftarrow \text{create_edge}(\text{doc}_{\text{src}}, \text{elem}_{\text{src}}, \text{block}_{\text{src}}, \text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{block}_{\text{dst}})$
 - 8 add edge_{new} to global set of temp. edges
 - 9 **create_hist_entry**(*“Create text”*, $\text{doc}_{\text{dst}}, \text{elem}_{\text{dst}}, \text{block}_{\text{dst}}, \text{subj}, \text{role}, \text{time}_{\text{curr}}$)
 - 10 $\text{pos}_{\text{insert}} \leftarrow \text{end of block}_{\text{dst}}$
-

are similar, we explain only one of them. Algorithm 8 is used to delete text. Since start or end of the text to be deleted can point into a block, the affected blocks have to be split (lines 1 and 2) to keep track of the deleted parts. Each deleted text block gets a final history entry to indicate its deletion (line 4). Finally, the text block is deleted (line 5).

Algorithm 8: Delete Text Content

Input : doc_{src} , elem_{src} , $\text{text}_{\text{start}}$, text_{end} , doc_{dst} , elem_{dst} , subj , role , $\text{time}_{\text{curr}}$

Output:

```

1 split_block( $\text{doc}_{\text{src}}$ ,  $\text{elem}_{\text{src}}$ ,  $\text{text}_{\text{start}}$ )
2 split_block( $\text{doc}_{\text{src}}$ ,  $\text{elem}_{\text{src}}$ ,  $\text{text}_{\text{end}}$ )
3 for each  $\text{block}_{\text{src}}$  within  $\text{text}_{\text{start}}$  and  $\text{text}_{\text{end}}$  of  $\text{elem}_{\text{src}}$  do
4   create_hist_entry(“Delete text”,  $\text{doc}_{\text{dst}}$ ,  $\text{elem}_{\text{dst}}$ ,  $\text{block}_{\text{dst}}$ ,  $\text{subj}$ ,
    $\text{role}$ ,  $\text{time}_{\text{curr}}$ )
5   delete_text_block( $\text{doc}_{\text{dst}}$ ,  $\text{elem}_{\text{dst}}$ ,  $\text{block}_{\text{dst}}$ )

```

5.2.3 Check-in

A user can activate the check-in command of the UI to save his changes to an opened document doc_a , which are currently stored only within the DP, to the Doc DB. As a result of this, the checked-in version of the document becomes relevant for the access decisions of other documents, which also includes concurrently opened versions of doc_a . For both kinds of affected documents the permissions must be recalculated, which possibly revokes permissions of currently edited documents. The concurrent editing of a document can also lead to conflicts, where the editing of one user to doc_a is incompatible to the editing of another user, who also has edited doc_a . For this reasons, we have to perform two steps when a document is checked-in. In step one, we have to resolve conflicts between the concurrent versions of a document. In step two, we must update the permissions of other affected documents whose permissions depend on the saved document.

To perform step one, we first retrieve the list of concurrently edited versions of doc_a , which is maintained by the Doc DB for each opened document. Next, we must merge all concurrently edited versions of doc_a to one consistent version. We must do this before we can evaluate the permissions for the document, because the permissions of an edited document can also depend on the content of the document itself. We apply a conflict resolution strategy to solve conflicts between concurrently edited documents. It depends on

the scenario to define a specific strategy. One possible strategy is to resolve conflicts manually. An automatic strategy can accept or reject changes depending on the role of the subject that performed the changes or depending on the time the changes were performed, since this information is available in the corresponding history. After the conflicts are solved, the temporarily stored edges, which correspond to the accepted operations, are saved to the Copy DB.

To perform step two, we first inspect the Copy DB to retrieve the opened documents that might depend on doc_a . These documents have at least one node, that is in is-copy-of relation with a node of doc_a . Then, we recalculate the permissions of these documents for their current users. In some cases, permissions of nodes that were edited in this moment are revoked. In these cases, the UI asks the user whether he wants to reject the current changes or keep them and accept being unable to make further changes.

5.3 Distributed System Architecture

So far, we have made no assumptions how the components of our system, which we have described in Section 5.1, are distributed across several physical machines. Since our system should be able to be used by a large number of users this assumption is not acceptable, since we must give a specific design how components are distributed to actually make the system usable. Moreover, we must define which components are executed on the machine of the user. We refer to this machine as the client machine or short as the client. As a consequence, we need to design a distributed system [CDK01] for the architecture that we have presented so far. In a distributed system, the components of the system are executed on physically different machines and communicate over a network with each other.

We will specify a distributed system architecture for three reasons. First, we want to illustrate that our model can be applied as a distributed system. Second, we will specify security mechanisms for our system architecture, which depend on the specific design of the system architecture as a distributed system, e.g., these mechanisms depend on whether components are implemented on a server or on a client and on whether components communicate with each other over a communication network or directly. Therefore, we need the design of our model as a distributed system as input for the specification of the security mechanisms. Third, we finally will implement our model as a distributed system, which requires the specification of the model as a distributed system as well. We will use the implementation for a performance evaluation.

We believe that our model can be implemented as a distributed system in many different ways and with different properties, e.g., different levels of security, complexity and efficiency. The way of implementing the model depends on the requirements of the scenario. In the following, we will investigate a possible approach to implement our model as a distributed system for scenarios with high requirements on security, e.g., a business scenario.

In the following sections, we will first decide which overall approach we will choose and then will we show different alternatives within that approach. We will argue which of these alternatives is best suited for the type of scenarios we described above.

5.3.1 Overall Approach

There are several different approaches to design a distributed system, such as the service oriented approach, the distributed shared memory approach, the computing grid or cluster approach, the peer-to-peer approach [Wal03, ATS04] and the client-server approach. The latter approach is also referred to as the classical approach for designing a distributed system. We need to choose one of these approaches for our distributed system architecture. We will focus on the peer-to-peer approach and the client-server approach, because we believe these approaches are best suited for our system. To decide which of these two approaches we will use, we first describe these approaches and their properties in terms of security, efficiency and availability.

Client-Server Approach

In the client-server approach, there are two different types of machines, namely clients and servers.

Servers are dedicated machines that perform services for the clients. These services either store data, e.g., a database, or perform computations, e.g., the evaluation of access control rules. Servers are dedicated machines on which no user is working. As a consequence, they can be placed in a secured area, e.g., a server room. Therefore servers are not physically exposed to the users of the system. Being less exposed makes servers a better choice for critical data or critical services.

The second type of machine is the client machine, which is the type of machine on which users are working. The data, which is sent to a client or stored on a client, has a much higher risk of being accessed in an unauthorized way, e.g., being manipulated, since the user of a client has more options to manipulate his machine compared to manipulating a remote server in a server room.

In addition to this, as servers can be used for only one dedicated purpose, their set of software can be much smaller compared to the set of software for a client, which usually must run a lot of different programs. A smaller set of software helps to make a system less prone to errors and less vulnerable to intentional attacks. Due to their smaller set of software and to being less exposed to the users of the system, servers offer a higher level of security for the software that is executed on them compared to software running on a client machine.

Nevertheless there are also drawbacks of using servers. If a service is implemented on a single physical server machine, the risk of losing the systems functionality is high. The crashing of this single server machine can render the complete system useless, if the services executes a function which is essential for the system. To increase the availability of the system, such services can be replicated over different physical servers. As a result, a service is executed on more than one physical server in a redundant way. The replication helps to increase the availability of the system, since in case that one server crashes there is a backup server for the functionality of the crashed server. Moreover, the replication of one service across multiple servers increases the maximum number of users the system can handle, since the total load of the service can be distributed to several physical machines.

Peer-to-Peer Approach

In the peer-to-peer approach, each client also acts as the server for other clients. These types of machines are referred to as peers. Some functionality, e.g., data storage or computation, is performed by peers instead of using a dedicated servers as described above. These peers have several drawbacks.

First, their availability is much lower compared to servers, since client machines are under the control of users, which can turn their machines off frequently, reboot them or take them to a different place with a different or with no network connection. As a consequence, a much higher level of redundancy is required compared to servers. For example, three redundant servers can be sufficient to increase the availability of a system since the probability of a simultaneous failure of all three servers is rather low. In contrast to this, distributing a critical service over three peers is completely unacceptable, since the chance of three peers being offline simultaneously is too high. In addition to the higher chance of a peer being unavailable, peers also enter and leave the system in a much higher frequency compared to servers, which only crash with a rather low probability. The reduced availability and the higher frequency of entering and leaving leads to much higher effort in keeping distributed data in a consistent state. In addition to

this, the risk of having outdated data is much higher in a peer-to-peer system, due to the higher frequency of peers entering and leaving the system. The last aspect is very critical for our system, since access control decisions must not be performed based on outdated data.

The second and more critical drawback for our regards is the much lower level of trust of peers, which are under the control of a user. In our system architecture, we have a lot of sensitive information and critical services, which should not be executed on a machine with a low level of trust. Although, the reduced level of trust can be compensated to some degree with additional security mechanisms, e.g., encryption and integrity measurement, the risk of data or services being manipulated by the user of a peer is still much higher compared to a server.

Choosing One Approach

Having the previous discussion in mind, we decide to design our system using the client-server approach, which has two advantages compared to the peer-to-peer approach. The first advantage is, that the complexity is much lower, since we need no mechanisms to secure the distributed storage of data, which otherwise could be easily manipulated by the user of a peer. Moreover, we do not need mechanisms to prevent the manipulation of computations performed on a peer. The second advantage is, that performing security relevant services on a dedicated server leads to a higher level of security, because servers have better properties in terms of security, e.g., servers must not be physically exposed to users, which might have an interest to manipulate that machine. Moreover, servers typically use a smaller set of software compared to client machines which are often used for a variety of different tasks. A smaller set of software usually has less vulnerabilities. Nevertheless, we are confident that our architecture can be implemented using the peer-to-peer approach.

5.3.2 Client-Server Approach

After we have decided that we want to use the client-server approach to design our architecture as a distributed system, there are still some open design decisions concerning the resulting architecture. For every component we must decide whether we want to implement it on the client side or on the server side. Obviously, we achieve the highest level of security if we implement all these components on the server. Nevertheless, we want to discuss the alternative combinations to show their individual benefits. Table 5.1 summarizes the components of our architecture and describes their function.

Component	Description
User Database	Stores user credentials and role hierarchy
Document Database	Stores documents
Copy Database	Stores is-copy-of relation between documents
Rule Database	Stores the access control rules
User Interface	Presents document and offer commands
Document Processor	Executes commands for editing a document
Policy Enforcement Point	Interrupts commands and queries PDP
Policy Decisions Point	Evaluates access control rules

Table 5.1: Components of our system architecture

As discussed above, we consider the client to be less secure than the server. Moreover, implementing one or more of the database on the client side, would give the system the character of a peer-to-peer system. As we discussed above, we do not want this. As a consequence, the four databases of our system should be implemented on the server side. The user interface must be implemented on the client side, since it is used to communicate with the user. The document processor can be implemented either on the client side or on the server side. If the document processor runs on the client, we must define that the document processor only operates on the documents of the corresponding user, to eliminate the risk of lost confidentiality. Since the policy enforcement point is tightly bound to the document processor it should be implemented on the same side as the document processor. The policy decision point can be implemented on the client side, too. In that case, we must define that it evaluates only rules concerning the corresponding user. Alternatively, the policy decision point can also be implemented on the server side. We summarize these options in Table 5.2.

If several components are implemented on the server side, it is not required that these components share one physical machine. Instead each component can be installed on a different machine and communicate with the other components using a communication network. Distributing components over several physical machines can help to improve the performance of the system.

Finally, we must decide on which side the document processor, the policy enforcement point and the policy decision point should be implemented. These decisions cannot be made independent of each other. Instead, we have to examine the dependencies between these components. For instance, the policy enforcement point is not a component itself, instead it is only a component in an abstract view of the system. In fact, the policy enforcement point is an an irremovable aspect of the document processor. As a conse-

Component	Side
User Database	Server
Document Database	Server
Copy Database	Server
Rule Database	Server
User Interface	Client
Document Processor	Server or Client
Policy Enforcement Point	Server or Client
Policy Decisions Point	Server or Client

Table 5.2: Possible sides to implement the components

quence, the policy enforcement point and the document processor are always implemented on the same side.

Combination 1: DP and PDP on the client

The first combination is to implement both document processor and policy decision point on the client side. This combination requires a trustworthy client, because the client calculates policy decisions for documents that the user of the platform wants to access. As a result, the interest to perform a manipulation is high. Moreover, the access control rules must be transferred to the client, because they are required to evaluate the rules. This requirement has two disadvantages. First, the rules can contain confidential information, which should not be exposed to the user of the client. For example, rules which deny the viewing of some top secret information can reveal the existence of that information, which might not be acceptable. To solve this problem, the rules can be sent encrypted to the client, but they must be decrypted to process them. While the rules are decrypted, the risk of information leakage is high. Second, this approach is not efficient in terms of storage and memory consumption. When the rules are transferred to the client, we cannot decide easily which rules might affect the documents of the client. As a consequence, we must send all rules to every client. As a result, the storage and memory requirement of the rules is multiplied by the number of clients of the system. In this combination, the client needs the computational resources to evaluate the rules. This increases the demand of resources on the client side, but reduces the requirements of the server.

Combination 2: DP on the client and PDP on the server

The second combination is to implement the document processor on the client and the policy decision point of the server. In this combination, the rules are not sent to the client, which removes two of the disadvantages discussed above. In this case, secret information in the rules is not revealed to the client and the clients do not need additional resources to store and process the rules. This combination has still a drawback concerning efficiency. The document processor and the policy decision point need access to the current document. The policy decision point requires access to the complete document to evaluate the rules, whereas the document processor only needs that part of the document that is accessible for the current user. The disadvantage of this approach is that these two version of the document must be synchronized over a communication network, because both versions reside on a different machine. As a result, a lot of the efficiency of this combination is lost, because the latency of the communication network seriously degrades performance. We illustrate the steps for performing an operation in Figure 5.2, where “S” is the abbreviation for the server and “C” is short for the client. The direction of the communication is indicated by an arrow.

1. C → S : Request to perform operation
2. S : Evaluate rules
3. C ← S : Send “allow” to client
4. C : Perform operation on censored document
5. C → S : Send censored document to server
6. S : Check whether client performed valid operation
7. S : Merge censored version with the uncensored version
8. S : Evaluate rules to calculate new censored version
9. C ← S : Send new censored version to client
10. C : Calculate new view of censored version
12. C : Present view to user

Figure 5.2: Protocol steps for performing an operation in Combination 2

In step one, the client requests to perform a specific operation on the document. Then, the server evaluates the rules to answer the request. In this example, we assume that the operation is allowed. Consequently, the server sends “allow” to the client. The client performs the operation on the document, which leads to a new version of his censored version of the document. We refer to the version of the document on the client as censored version, because some nodes might be removed due to missing permissions to view them. In contrast to the client, the server keeps the full version of the

document, which includes the nodes the client is not allowed to see. Next, the client sends the updated censored version to the server. The server checks whether the client performed a valid operation. Next, the server merges the censored version of the document with the full version of the document and then evaluates the rules to calculate a new view for the client. After that, the server sends the new censored version to the client. The client calculates a view of the censored version, which can have a different representation as the internal one, e.g., a graphical representation or a text representation. Finally, the UI on the client side presents the view to the user.

We have described this process to illustrate that the implementation of document processor and policy decision point on different side leads to a significant overhead. We will show that these disadvantages can be avoided by using Combination 4, which implements both components on the server. But first, for the sake of completeness, we will discuss Combination 3, which is the least favorable one.

Combination 3: DP on server and PDP on the client

The third combination is to implement the document processor on the server and the policy decision point on the client. This combination combines the disadvantages of having the policy decision point on the client side with the disadvantages of implementing document processor and policy decision point on different machines. As a consequence, this combination is the least preferable one.

Combination 4: DP and PDP on the server

In Combination 4, we implement both document processor and policy decision point on the server, which has two advantages. First, it removes the overhead of performing the communication between document processor and policy decision point over a communication network as we illustrated in Combination 2. Second, having both components on the server side, leads to the highest level of security, since all security relevant components are implemented on the server. As we argued above, we consider that the server has a much higher protection level than the client. To illustrate the benefits of avoiding the overhead of Combination 2 and to describe the final protocol, we depict the protocol steps to perform an operation in Combination 4 in Figure 5.3.

The first two steps in this combination are identical to the protocol steps of the protocol of Combination 2. In step three, we can avoid communication overhead by not having to send the result of the rule evaluation back to the

1. C \rightarrow S : Request to perform operation
2. S : Evaluate rules
3. S : Perform operation on uncensored version
4. S : Evaluate rules to calculate new censored version
5. S : Calculate new view of censored version
6. C \leftarrow S : Send view to client
7. C : Present view to user

Figure 5.3: Protocol steps for performing an operation in Combination 4

client. As above, we assume that the result is to allow the requested operation. In step three, we can avoid to perform the operation on the censored version and to merge this version with the uncensored version, because the server is allowed to operate on the uncensored version. After this, we evaluate the rules to retrieve a censored version of the document and to create a view for the current user for it. If we do this on the server, we can also perform the view creation directly in one step. Finally, the view is sent to the client and presented to the user. In total, we have saved communication overhead by reducing the communication steps by half and to avoid at least 4 steps in the protocol. On the downside, this combination leads to a high amount of computations on the server. We will show in the performance evaluation in Chapter 7 that the resource requirements for the server are acceptable.

Comparison of the combinations

It depends on the requirements of the scenario to choose one of the four combinations. We have summarized the properties of these combinations in Table 5.3. In this table, we have listed each combination in a different row and have rated its level of security and its efficiency. The second and third column indicate on which side the document processor and the policy decision point is implemented. Additionally, we have listed the requirements on the server and on the client. These requirements include both the level of security as well as the amount of resources the corresponding machine should have, e.g., high requirements on the client mean that the client should have a high amount of resources and a high level of security.

Implementing the policy decision point on the client introduces high risks of compromising the confidentiality of documents of the system and the rules. To evaluate the rules for one document the policy decision point must also access the documents which contain nodes that are in is-copy-relation with node of the currently processed document. As a consequence, a successful attack on the policy decision point on the client can lead to unlimited access to

	DP	PDP	Security	Efficiency	Requ. Client	Requ. Server
C1	C	C	\ominus	\oplus	high	low
C2	C	S	\oplus	\ominus	medium	medium
C3	S	C	\ominus	\ominus	high	medium
C4	S	S	\oplus	\oplus	low	high

Table 5.3: Comparison of the four combinations

all documents of the system. Moreover, the policy decision point must access all rules of the system which also can contain confidential information, e.g., the existence of some top secret information might be revealed by inspecting the rules. As a result, we regard all combinations where the policy decision point is implemented on the client as having a higher risk of manipulations.

We choose Combination 4 as basis for our security architecture and for our implementation, because its implementation of critical services on the server makes is robust against manipulations. Moreover, the protocol for performing an operation can be designed in a way that makes the protocol very efficient. In scenarios where the security requirements are low and servers with high performance are not available Combination 1 is a good choice. Combination 1 is only acceptable if client machines with a very high protection level are used. These clients must be setup in way where the risk of manipulations is very low. Nevertheless, the risk of losing the confidentiality of all documents as a result of a successful attack on a client is still present.

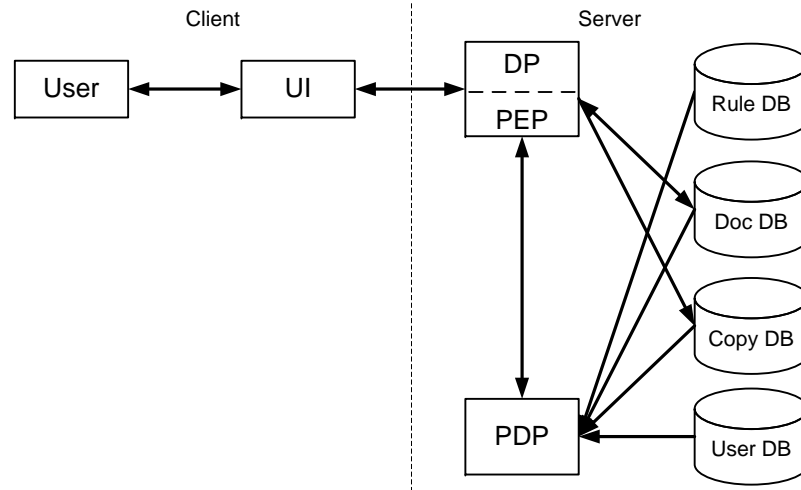


Figure 5.4: Distributed System architecture

The final distributed system architecture is presented in Figure 5.4. In

this architecture, all components except of the user interface are implemented on the server side. Since the server has to handle many performance intensive tasks like rule evaluation, we have the risk that the system does not scale well with the number of documents or the number of clients. Therefore, we conclude this section with a discussion on the scalability of our distributed system architecture.

5.3.3 The scalability of the distributed system architecture

As mentioned before, the components of the server are not required be implemented on a single physical machine. Instead, every component can be set up on a dedicated machine. Each component of the server can also be set up by combining multiple machines to perform the service of one component. In this case, each component exists redundantly several times, where each instance is located on an individual machine. This technique is used to increase both performance and availability. The workload for the component in question must be distributed to the individual instances of the component. This distribution process is referred to as load balancing.

Concerning the four databases of our system, there has been a lot of research on increasing the performance of databases and making them more scalable, e.g., [DR92, PSTT96, JPPMKA02, Tal03]. The techniques to achieve these goals are query optimization, replication and efficient mechanisms to keep the replicas in a consistent state. Query optimization is about to find a semantically identical expression of a query which can be executed in the shortest possible time compared to the original expression of the query. Databases can be replicated across several physical machines to increase the performance of read operations. On the downside, these so-called replicas must be kept in a consistent state, when they are updated by a write operation. Replication increases also the availability of a system, since if one of the replicas becomes unavailable, the other replicas can still be used. This also reduces the problem of a single point of failure. Since, the replication techniques for databases are a well known research topic and since the replication of the databases does not affect our architecture design, we do not focus on this topic and view each database as a single component. If replication is required, it can be applied any time without the need to change the architecture.

The replication technique can also be applied to the remaining four components of the server. Since these components only perform computations and store their data in databases, replication can be done with less com-

plexity, since no mechanisms are required to keep the replicas in a consistent state. Nevertheless, we also do not replicate our components. If this should be required, we are confident that these components can be replicated without adding to much complexity to the architecture, since these components are only performing computations and access the data in the databases.

Now, we have defined a distributed system architecture. In the next chapter, we will analyze this architecture and investigate which security mechanisms are required to avoid potential threats.

Chapter 6

Security Architecture

In this chapter, we will design the security architecture for the system architecture that we have proposed in the previous chapter. For this purpose, in Section 6.1, we analyze potential threats on our architecture. After this, we present the security mechanisms that avoid or reduce the risks described in the risk analysis in Section 6.2. Finally, we evaluate the security mechanisms and discuss whether they achieve their goals in Section 6.3.

6.1 Risk Analysis

In this section, we evaluate possible attacks on the components of our architecture. Attacks in general can aim at different goals, e.g., at stealing data or at performing sabotage. We present five of these goals, explain them in general and give an example of what types of attacks are possible on our architecture.

Confidentiality: A system ensures the confidentiality if only authorized subjects are able to access protected data. Within our architecture, we must ensure the confidentiality of the documents and the confidentiality of the rules. Therefore, we must ensure that the documents and the rules are not accessed by an unauthorized subject.

Integrity: Integrity defines that data is not manipulated by an unauthorized subject without detecting the manipulation. If possible the manipulation of data by unauthorized subject can be prevented at all. Within our architecture, we must ensure that neither the documents nor the rules are manipulated by an unauthorized subject.

Availability: The availability of a system is given, if the system is able to execute its intended function in a timely acceptable way. If an attacker

is able to manipulate a system to operate very slow or to be non-functional then the availability of the system is compromised.

Authenticity: Authenticity defines that the identity of a subject or an object is not forged.

Non-repudiation: Non-repudiation defines that a user must not be able to deny having performed an action that he has performed.

In addition to that, an attack can have different targets and can be performed in different modes. Theoretically, each of the components of our system architecture (see Figure 5.4) can be the target of an attack, but since only the document processor must be reachable for the clients, we assume that for a successful attack this component must be attacked.

An attack can be either performed locally or remotely. In a local attack, the attacker has physical access to the attacked machine and has a user account on the corresponding machine. For example this type of attacker can access the local file system or reboot the machine with another operating system. In contrast to this, in a remote attack, the attacker uses a communication network to attack his target. The only options for a remote attack are to use the communication channels of our architecture. We further assume that an attacker has no physical access to server components, which we think is realistic, since servers can be protected by placing them in a secured server room. As a result, local attacks can be performed only on the client machine. Moreover, we assume that the administrator of the server is trustworthy.

Next, we will describe two different types of attackers, namely the inside attacker and the outside attacker. Each type of attacker has certain characteristics that we will describe in the following.

Inside Attacker: An insider attacker is a legitimate user of the system who misuses his permissions to perform an attack. We assume that an inside attacker also has physical access to the client machine, which enables additional attacks. In addition to this, an inside attacker can perform remote attacks on the document processor. For example, if our system is used within a company then an inside attacker is typically an employee of the company.

Outside Attacker: An outside attacker is a user who does not have legitimate access to the system, which means that he has no credentials to login into the server and also has no physical access to any machine. The outside attacker can only perform remote attacks on the client and the document processor. For example, if our system is used within a

company then an outside attacker could be either a professional hacker hired by a competitor, a spy performing industry espionage or a hacker who tries to perform attacks just for his own entertainment.

An inside attacker can perform local attacks on his client machine on which the user interface is executed. In addition to that, an inside attacker can also perform remote attacks on the document processor and on a user interface, which runs on another client machine. As a consequence, an inside attacker can perform four types of attacks according to this classification scheme. An outside attacker can only perform remote attacks on the user interface and on the document processor. All together, an inside attacker can perform every attack an outside attacker can perform plus an additional attack, which only he can do. Therefore, we believe it is sufficient to inspect the attacks which can be performed by an inside attacker, since this type of attacker has the most options for an attack. We will start with a discussion of possible attacks on the client.

6.1.1 Attacks on the Client Machine

In this section, we describe possible attacks in the scenario mentioned above. We focus on attacks that compromise the confidentiality of the protected data. Moreover, we organize the attacks in groups, where all attacks of the group have a common method of attacking the client.

Software Manipulations.

In the following, we discuss software manipulations on the client, which can be performed on different components of the system.

Extract data from the user interface. An inside attacker can try to extract confidential information from the user interface. As a result, the user interface must be designed in a way, that it is not possible to extract confidential data from it and transfer it to another application, e.g., an e-mail client.

Manipulate the user interface. If the user interface prevents the extraction of confidential data, the attacker can try to manipulate the corresponding protection mechanisms. Alternatively, the attacker could use another user interface that is compatible with the protocols used in our system architecture, but allows data to be extracted. Therefore, we must have a mechanism to ensure that the user interface is not modified and that it is the version that has been deployed originally.

Use the operating system mechanisms to extract data. The attacker could also use the underlying operating system on which the user interface runs to

extract data. One such method is to extract the confidential data from the memory by writing the memory used by the user interface to a file, which is also referred to as a memory dump. An attack like this requires that the application of the attacker runs in kernel mode, which enables access to the entire physical memory of the machine. There are many other similar methods, which all have in common that they use the services offered by the underlying operating system. Consequently, we must configure the underlying operating system in a way that prevents the use of services like memory dumps to extract the confidential data from the user interface.

Manipulate operating system. If the underlying operating system is configured in a way that prevents the extraction of confidential data using its services, the attacker could modify the operating system or its configuration. The attacker could try to re-enable the mechanisms that we have disabled before. For example, he could exchange a system module, e.g., the module that performs memory management or the module that displays data on the screen. Another similar approach is to replace the entire operating system with a system that allows the extraction of data. As a result, we need a mechanism to ensure that the operating system is neither manipulated nor entirely replaced. This mechanism must ensure that the operating system is authentic, which means that it is the one that has been deployed and it is configured as we have defined.

Extract confidential data from the swap file. Our system architecture does not store documents permanently on the client. Instead, our system architecture keeps them in the memory to display and edit them, which avoids attacks based on accessing the hard drive of a client machine. Nevertheless, we must also ensure that the operating system does not swap the memory used by the user interface to the hard disk, if available memory is getting low. This again must be done by configuring the operating system. As a consequence, there is no risk of losing confidential data by stealing a hard drive of an authorized client.

Masquerading Attacks.

In the following, we discuss different types of masquerading attacks on our system architecture. We describe the cloning of a client machine and the spoofing of the server of our system architecture.

Clone a client machine. Another type of attack is to clone an authorized client machine by creating an exact copy of the configuration of an existing machine, e.g., by creating an exact copy of the hard disk of the authentic client or by copying the authentication credentials to another client. For example, a legitimate user could set up such a cloned client machine in an

area which is not under surveillance to extract data by taking pictures of the displayed data. To reduce the risk of this type of attack, we need a mechanisms that prevents the cloning of a client machine. This mechanism must ensure that the identity of the client machine is bound to the hardware of the machine.

Masquerade the server. The attacker can also masquerade the server, e.g., by redirecting the network traffic using a DNS poisoning attack. Therefore, we need mutual authentication between client and server. In our scenario, the masquerading of a server is less dangerous, since no confidential documents are uploaded to the server. However, this can be the case in the scenario of corporate computing on home computers.

Hardware Attacks and Analogue Attacks

In the following, we discuss attacks on the hardware and analogue attacks. We consider Direct Memory Access (DMA) attacks as hardware attacks and discuss them in this section, too.

Use DMA to extract data. Another type of attack is to use a device with DMA to extract confidential data. DMA bypasses any protection managed by the CPU and allows to access the entire memory.

Extract data using probing attacks. Moreover, the attacker can perform probing attacks, such as mechanical or electrical probing attacks, on the hardware components of our system architecture. Using this attack, he can extract confidential data directly from the hardware, e.g., from the TPM or from a memory module.

Use analogue channels to extract data. Besides the attacks mentioned above, the attacker can take an analogous screen shot of the display using a camera.

All attacks discussed so far are summarized in Table 6.1 together with the mechanisms that are required to prevent the corresponding attack.

Attack	Required mechanism
Manipulate UI	Authentication mechanism for UI
Spoof the server	Mutual authentication
Use OS to extract data from UI	Configure OS to prevent this
Manipulate OS	Authentication mechanism for OS
DMA attack	Countermeasures against DMA
Extract data from swap space	Configure OS not to swap UI
Clone a client	Prevent cloning of clients

Table 6.1: Attacks on the client and required mechanisms

6.1.2 Attacks on the Server

Since only the document processor must be reachable for clients, we assume that only this component of the server side can be attacked. Moreover, we assume that the server components run on dedicated machines and no user is working locally at such a machine. Thus, the server can be attacked only remotely using the communication network. We further assume, that the server only runs the services of our architecture and no other services. As a result, the only way to attack the document processor is using the interface that it provides for clients. As mentioned in the previous section, we require that both the client machine and its user is authenticated with an authentication mechanism. Thus, we do not regard spoofing of clients in this section.

The attacker can use the interface of the document processor to issue commands that put the document processor in an unwanted state, e.g., a crash. These types of attacks exploit flaws of the design or implementation of the communication protocol. For example, an attacker could send a command with a parameter outside the specified range. If such an error is not handled correctly, it could result in a crash of the corresponding component. In this case, the attacks aims at the availability of the system. A similar, but more dangerous type of this attack is to send a parameter consisting of more data than specified in the communication protocol. If the protocol is implemented careless, then the oversized parameter can exceed the size of the buffer where the data is written to and overwrite data on the stack of the executing program. Therefore, this attack is referred to as *buffer overflow attack* [CWP⁺00, LE01]. This attack can be used to inject executable code in the binary of the attacked program. This is achieved by overwriting the return address of the current procedure, because this return address is stored on the stack too. The new return address points to injected code that is also part of the oversized parameter. When the current procedures exits and control flow should be passed to the calling procedure the overwritten return address is read from the stack and jumped to. At this point the injected code of the attacker is executed. For example, this code can open a remote login shell. To be able to perform a successful buffer overflow attack, the attacked program must be implemented careless by not checking the size of an input parameter. Using this attack method, the attacker can gain the permissions of the process that he has attacked. In our case, the attacker could take control over the document processor, which is able to load and store documents from the document database. Hence, we must implement our architecture in a way that it is not vulnerable to buffer overflows and maliciously formatted input parameters.

Another type of attack aims at the availability of the server by causing an overload on one component of its components, e.g., by issuing commands in a high frequency. For example, a legitimate user could request views of different documents several times per second. Since the server must evaluate the rules to calculate a view, a lot of computational resources of the server are used for that process. Such an attack can increase the time the server needs to execute commands for other clients. If this time exceeds a certain threshold, the performance of the server is not more acceptable and thus it cannot be used by the other clients. Therefore, such an attack is referred to as a *deny of service (DoS) attack* [Nee94]. Since only authorized users are able to execute all commands of the server, e.g., the calculation of a view, this type of user has the most potential to perform a DoS attack. Unauthorized users can try to perform DoS attacks by performing authentication requests in a high frequency. DoS attacks are difficult to avoid since they use legitimate commands. One approach to avoid vulnerabilities against DoS attacks is to limit the frequency of certain resource-intensive commands. For example, we can define that at most one view per second can be calculate for a single authorized user. In this thesis, we will only inspect vulnerabilities against DoS attacks that are introduced by our components. There are many other DoS attacks, e.g., attacks that cause a high network traffic, that are not special for our architecture and therefore discussed in the literature, e.g., [SKK⁺97, Mea99, KK03].

Additionally, an attacker can eavesdrop the communication between the server and the client. Such an attack can aim both at the confidentiality of the transferred documents and on the authenticity of client and server, because the attacker can retrieve either the documents or the user credentials from the transferred data. To prevent this attack, we must encrypt the communication between client and server, which is a common method to achieve confidentiality.

Moreover, a legitimate user could edit a document and deny having done so later. For example, a user could delete important conditions of a contract or change the amount of money granted to a project in the corresponding document. We achieve the non-repudiation of performed operations by logging them in the history. As a result, we do not need an additional mechanism to achieve this security goal.

All attacks on the server discussed so far are summarized in Table 6.2 together with the mechanisms that are required to prevent the corresponding attack.

Attack	Required mechanism
Bad input parameter	Careful implementation
DoS attack	Limit rate of certain commands
Eavesdrop network communication	Encrypt communication
Repudiation of performed operations	handled by the histories

Table 6.2: Attacks on the server and required mechanisms

6.1.3 Requirements for the Security Architecture

In this section, we present the requirements for the security architecture. The first two requirements apply to any scenario, whereas the requirements that follow are specific for our system architecture.

Requirement 1: *The operating system is configured to support only the minimum set of services and resources.* As we have discussed so far, some of the mechanisms for securing the client must be performed by configuring the operating system, e.g., configuring it to not swap the memory occupied by the user interface or disabling the possibility to take digital screen shots. Moreover, we must configure the operating system to allow only network connections to the document processor, which reduces the risk that data is sent to an unauthorized third party.

Requirement 2: *The number of software components of the operating system is minimal.* To reduce the risk of vulnerabilities, it is highly desirable to employ an operating system that has the minimal set of functions that are required to execute the user interface, because the chance of vulnerabilities increases with the complexity of a system. Consequently, we must use a operating system with the minimal number of user space software and kernel components, e.g., device drivers. For example, we can use a microkernel-based operating system and remove not required components from it.

The downside of this approach is that the client machine might be rendered unusable for other software that should be executed on it, because the operating system that we have installed can be incompatible with this other software. In the case when the client machine is used to execute only the user interface this is no problem. But in the other cases, which we think are very likely, the user either needs a second machine or must reboot the machine to use the operating system that is compatible with the other applications that he likes to run.

Requirement 3: *Attestation of the client machine.* Moreover, we need

to ensure that the operating system that we have supplied is neither exchanged with another operating system nor is its configuration manipulated to disable the security mechanisms of the security architecture. For that purpose, we will use the mechanisms defined by the Trusted Computing Group (TCG) and that we have described in Section 2.3. The remote attestation is a mechanism that allows a remote platform, e.g., the server, to check whether the system is in the correct state.

Requirement 4: *Authenticity of client and server.* The user of the client platform could manipulate his machine to forward an attestation request to another machine with authentic software. Therefore we must ensure that the provided configuration values refer to the attested system and that the authenticity of both client and server is guaranteed.

Requirement 5: *Completeness of attestation.* The definition of this state depends on what is measured on the client side. The usual approach [SZJvD04] is to calculate the hash value of every executed binary and include it in the state definition. As a result, the verifier receives a list of all binaries that have been executed since the last reboot of the machine. However, this approach has two problems. First, the received list of executed binaries gives no hint about what was executed after the remote attestation. For example, a key logger or trojan horse can be started after the remote attestation. Although the malware was located on the client machine at the time of the remote attestation, it was not detected, because it was executed after the remote attestation. The second problem is, that using this approach, only the executable binaries of applications are inspected. Shell scripts and configuration files are not included in the measurement. This leads to the problem, that certain manipulations, e.g., manipulations of the configuration files, cannot be detected by this approach. Consequently, we need an approach that measures the complete system configuration including scripts and configuration files.

6.2 Security Mechanisms

As discussed before, we must ensure that the client machine can execute additional software, e.g., a web browser or an e-mail client, after the security mechanisms are applied. This can be a problem when requirements 1 and 2 are fulfilled, since they reduce the compatibility of the client machine with additional software. For this reason, it is desirable to have a different operating system for different applications. As a consequence, we need a method

to run different operating systems on one client machine, which we achieve by using virtualization techniques and running different virtual machines on a client machine. Each operating system can be executed in a separate virtual machine. Concerning the security of the system, we must ensure that these virtual machines cannot influence each other. For example, we must ensure that malware running in one virtual machine cannot extract confidential data from another virtual machine. We apply an approach for providing different virtual machines [SBHE07] and analyze whether the attacks described in Section 6.1 can be solved with this approach. The strong isolation achieved through virtualization guarantees that different virtual machines cannot influence each other. The approach [SBHE07], on which we base our architecture, uses virtualization in combination with the mechanisms defined by the Trusted Computing Group. It establishes several different execution environments by using various types of virtual machines, which are strongly isolated from each other. It also provides an abstraction of the underlying hardware TPM through a virtualized TPM (vTPM) interface. This allows the different virtual machines to use the measurement and reporting facilities of the TPM, thus they benefit from a hardware-based trust anchor. This approach has the advantage over the others [Bas06] that the binding between TPM and vTPM is already specified, which is useful for remote attestation of virtual machines.

We use this approach to execute the user interface in one such isolated virtual machine. Other applications are executed in a different virtual machine. Thus, they cannot interfere with the user interface. Figure 6.1 depicts the approach applied to our system architecture.

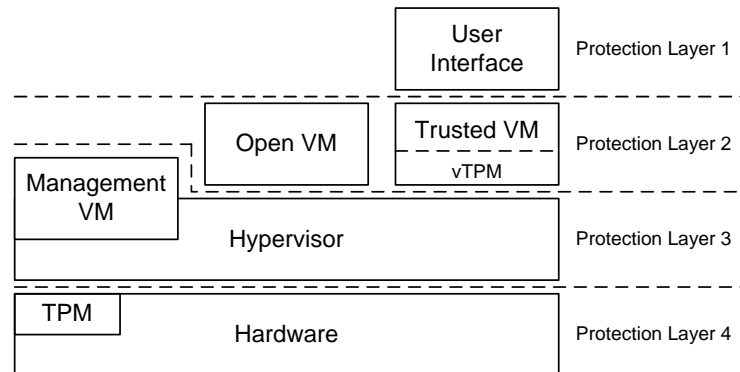


Figure 6.1: Security architecture organized in layers

The resulting security architecture consists of components divided into four protection layers, in which components located on one layer provide

security mechanisms to protect the components located on the layer directly above. In the case of a successful attack on one layer, the layers below can prevent the attacker from successfully transferring data to another physical machine. The layers do this by using additional mechanisms that ensure that data cannot be transferred elsewhere, e.g., by restricting network connections to the server of the architecture.

These components include three virtual machines, namely the *open virtual machine*, a *management virtual machine* and the *trusted virtual machine* (TVM). Additionally, a hypervisor partitions the underlying hardware and a TPM serves as a hardware-based trust anchor. The TPM provides hardware-based tamper-evident cryptographic functions to protect the software components running on the layer directly above from unauthorized manipulations. Together with remaining hardware components, the TPM forms the lowest protection layer, more specifically, protection layer 4.

6.2.1 Protection Layer 4: TPM and Hardware

The TPM is the anchor of trust and the basis for the attestation. We store several non-migratable client-specific keys in the TPM, which are used for the challenge-response authentication with the server. This prevents the cloning of a client machine and additional attacks on the authenticity of the client machine. The attacker must perform a local physical attack, e.g., a mechanical or electrical probing attack, to the TPM to extract these keys. Since the TPM is specified to be tamper-evident, these attacks are not prevented, but can be detected afterwards. We assume that the hardware and the TPM behave as specified.

6.2.2 Protection Layer 3: Hypervisor and Management VM

The hypervisor is the first part of protection layer 3 and provides an abstraction layer to the underlying hardware. It has privileged access to the hardware and can grant and revoke resources, e.g., CPU cycles, to and from the running VMs. This hypervisor provides strong isolation of the virtual machines, which is the protection mechanism of this layer. It ensures that different virtual machines cannot influence each other, e.g., by reading each others memory. In this approach, every virtual machine uses individual virtualized device drivers, which are executed within that VM. The hypervisor ensures, that these device drivers can only access the memory of the corresponding virtual machine. When running applications of different trust levels

on a machine without virtualization, an attacker could use a malicious device driver to gain system wide access, e.g., to read the memory of the user interface and extract the confidential documents. We assume that the hypervisor and the management VM are set up by a trustworthy system administrator and that the user of the machine is not able to change this configuration.

Because of its privileged position, the hypervisor needs to be trustworthy, since it can manipulate the CPU instructions of every virtual machine. We assume that the hypervisor is trustworthy and therefore guarantees strong isolation. Currently available virtualization solutions provide strong isolation. However, this still can be circumvented with direct memory access (DMA) operations [FHN⁺04]. These operations access the memory without intervention by the CPU and therefore bypass the hypervisor's protection mechanisms. Hypervisors with *secure sharing* [KZB⁺91] prevent these attacks, but suffer from a high performance overhead, as well as a large trusted computing base, since the required I/O emulation is moved into the hypervisor layer.

The management virtual machine is the second part of protection layer 3. It is responsible for starting, stopping and configuring the virtual machines. It is part of this protection layer, since it is closely connected to the hypervisor and is a privileged VM, which has direct access to the hardware TPM.

6.2.3 Protection Layer 2: Open VM and Trusted VM

This protection layer consists of the open VM and the trusted VM. The open VM is allowed to run arbitrary software components. It runs applications with a lower trust level, such as web browsers or office applications. The open VM provides the semantics of today's open machines and therefore has no additional protection mechanisms for upper layers. Since this virtual machine is not of interest for our approach, we will not focus on it in the rest of our work.

The TVM runs the user interface and a tiny OS with a minimal number of software components, to reduce the possible number of security vulnerabilities. This fulfills requirements 1 and 2 without losing the ability to run other software on the client machine. The tiny OS and the user interface are part of a virtual appliance (VA), which is a fully pre-installed and pre-configured set of software for virtual machines. To ensure that the VA is not manipulated, the management VM measures its integrity before startup.

The TVM runs in protection layer 2 and provides a virtual TPM (vTPM) as an additional protection mechanism. The operating system of the TVM uses this vTPM to protect the user interface running in protection layer 1. We use this vTPM to perform a complete attestation of the entire hard disk

of the TVM, to ensure that neither the operating system, its configuration, nor the user interface running on top of the operating system is manipulated. As a result, the verification of the state of the entire virtual machine requires only one reference value. This eliminates the need to maintain a large amount of reference values, which is the main disadvantage of the binary attestation. Moreover, the server checks in the attestation whether the protection layers below the operating system, e.g., the hypervisor, are trustworthy. As a result, this mechanism fulfills Requirement 5. In addition to that, we use the vTPM to establish an authenticated channel between client and server. We discuss the corresponding protocol in Section 6.2.5.

The TVM only accepts network requests from the server to reduce the chance of network attacks. After the booting process, which cannot be interrupted, the operating system of the TVM directly executes the user interface. As a consequence, the only option for a user to interact with the TVM is to use the user interface provided by us. This improves the security of the TVM, since it limits the number of possible attacks.

All I/O interfaces which can be used to extract data from the system are either blocked or controlled. For example, the management VM ensures that the hard disk is read only, which prevents an attacker from temporarily storing confidential data on this disk to extract it afterwards by booting a different operating system. Additionally, the management VM has a configuration file for the TVM, which defines that network connections are only allowed to the server of our system architecture, which inhibits an attacker from sending confidential data to a different host. As a consequence, even if an attacker exploits a vulnerability of the user interface, he cannot transfer the confidential data out of the TVM.

6.2.4 Protection Layer 1: User interface

This protection layer consists of the user interface. The user interface is written in Java, which is expected to minimize the risk of buffer overflows. In addition to this, the user interface can edit confidential documents in memory and does not need to write them to disk. The server's authenticity is checked by verifying the server's certificate before answering an attestation request.

6.2.5 Attestation Protocol

To prevent masquerading attacks on the authenticity of the platform configuration, we use an enhanced remote attestation protocol [STRE06]. These

attacks forward the integrity measurements of a conform host to masquerade a conform client state. The enhanced protocol adds a key establishment phase, to ensure that the channel of attestation is authentic. It also guarantees an end-to-end communication and prevents the attestation channel from becoming compromised by another application which could take over the attestation channel after the attestation has succeeded.

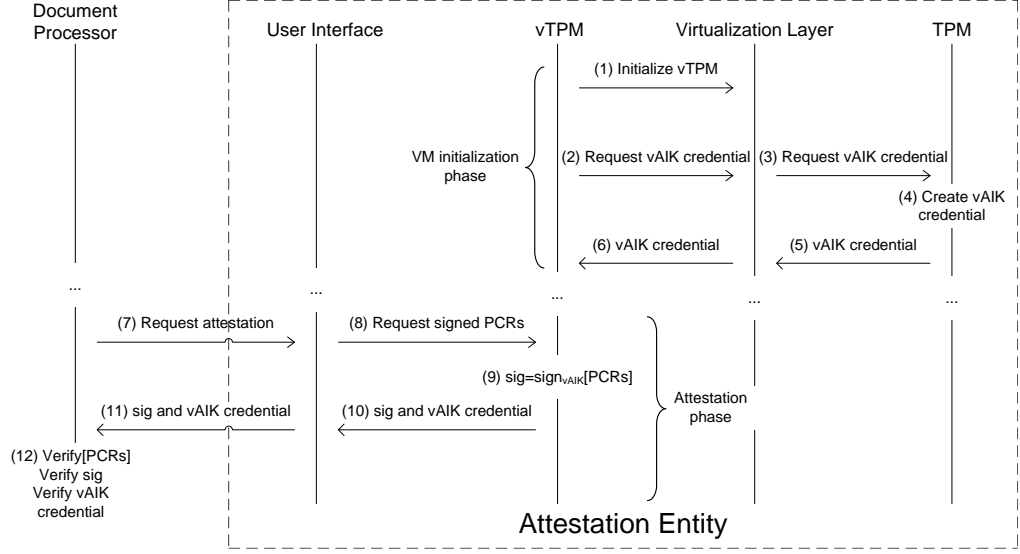


Figure 6.2: Attestation protocol

The protocol of our remote attestation is illustrated in Figure 6.2. It consists of an initialization phase and an attestation phase. The initialization phase yields a vAIK credential which is then used in attestation phase to sign the PCRs. This vAIK credential is signed by an AIK from the hardware TPM. In the first step of the initialization phase, the vTPM is initialized, which in turn requests a new vAIK credential from the hardware TPM (steps 2 and 3). The hardware TPM issues a vAIK credential and sends it to the vTPM (steps 5 and 6). The attestation phase is triggered by the document processor, which sends an attestation request consisting of a *nonce* and its public Diffie-Hellman key pair $g^a \bmod m$ for the key-establishment to the user interface (step 7). The user interface generates the corresponding Diffie-Hellman key pair $g^b \bmod m$ and sends this, together with the *nonce*, to the vTPM (step 8). The vTPM generates a digital signature using the vAIK (step 9) and transfers it together with the vAIK credential, to the user interface (step 10). The user interface forwards this data to the DP (step 11). Next, the DP verifies the authenticity of the user interface and its platform by inspecting the platform configuration registers (step 12). Finally, both the

DP and the user interface calculate the shared session key for the following communication (step 13, not illustrated).

The protocol for performing remote attestations guarantees that the endpoint of the communication is within the attested virtual machine. Therefore, relay or masquerading attacks are not possible. The combination of attestation and key establishment fulfills Requirement 4, since it prevents the masquerading of a trustworthy system configuration.

6.3 Evaluation of the Security Mechanisms

In the following section, we evaluate whether the security mechanisms described in Section 6.2 prevent the attacks mentioned in Section 6.1.

Software Manipulations.

Each of the protection layers can be either manipulated during runtime or before it is executed. Runtime attacks are especially critical, since they are not detected by the current method of integrity measurement, which only measures components when they are executed. The underlying protection layer must be manipulated to modify the current layer, because the integrity of each layer is measured before execution by the layer below. This results in a chain of trust, with the TPM as a hardware anchor. As a consequence, the manipulation of any layer either requires a runtime attack or a hardware attack on the TPM. In the following, we discuss possible runtime attacks on each layer. The user interface is robust against buffer overflow attacks, since it is written in Java, which is commonly believed to decrease the possibility of buffer overflows. On the downside, other attacks, such as exploiting other programming errors, are still possible. The risk of runtime attacks on the operating system is reduced, because software with lower complexity is expected to have less errors than software with higher complexity. Moreover, we can choose a strict system configuration to minimize possible attack methods, e.g., network connections are restricted to the server, the hard disk of the TVM is read-only and swapping is disabled. As a consequence, in the case of a successful attack on the operating system, the attacker has no options to transfer confidential data to another machine. This is an example of a lower layer preventing a successful attack, when the protection mechanisms of the layer above failed. Runtime attacks on the hypervisor are difficult, since it has a lower complexity compared to operating systems and it does not expose interfaces to the user which could be used for an attack. The management VM offers no interface either. Moving protection mechanisms

of upper layers into this layer, simplifies the verification of the correctness of these mechanisms due to the smaller size of this layer. A critical example of a runtime attack on the management VM is to masquerade an authentic system administrator of the management VM, e.g., by guessing the corresponding password. After that, he can perform an attack, e.g., modify the integrity measurement of the TVM to masquerade a trustworthy TVM.

Masquerading attacks.

Both client and server can be masqueraded, where the cloning of a machine is a special type of this attack. Since we assume that the server is trustworthy, we only focus on cloning attacks on the client. An attacker can create an exact copy of a client's hard drive, but he cannot copy the content of the corresponding TPM. The attacker cannot use this cloned client to access the server, since our attestation protocol uses secrets stored in the TPM and therefore detects that the client is not authentic. One such secret is the AIK of the hardware TPM, which is used to sign the vAIK of the vTPM. This vAIK in turn, is used in the attestation protocol. The server can check the authenticity of the vAIK with the corresponding AIK credential that was installed on the server when the system was set up. Using an honest system to masquerade a trustworthy system state is prevented by our attestation protocol. This protocol also detects and prevents masquerading of a server, since the server's certificate is checked by the user interface.

Hardware attacks and analogue attacks.

DMA attacks can either be handled entirely in software by emulating all I/O devices, which causes a high performance degradation. Alternatively, DMA attacks can be prevented by using hardware support, e.g., Intel's Trusted Execution Technology. As a consequence, it depends on the implementation of our security architecture whether or not DMA attacks are possible. Probing attacks are difficult to inhibit. At least the TPM is specified to be tamper-evident, which allows an attack to be detected afterwards. Analogue attacks are difficult to prevent with software mechanisms. Fortunately, the bandwidth of this channel is much lower compared to digitally copying confidential data. In addition to that, this attack method has a higher risk of being detected, if the machine is located at a monitored location, e.g., an office with many co-workers or an office that is under surveillance by security cameras. We do not provide a mechanism to prevent this type of attack.

Chapter 7

Implementation

In this chapter, we will describe the implementation of the prototype of this thesis. This prototype implements the system architecture described in Chapter 6. The security mechanisms, which we presented in Chapter 6, are not implemented in this prototype. These mechanisms are implemented in [RSGE06], [STRE06] and [SBHE07]. The implementations of these mechanisms demonstrate that the described mechanism are feasible. In addition to this, the current prototype of this thesis can be extended with the mechanisms described in Chapter 6. Since, we do not plan to do a further analysis on the combined implementation of the model together with its security architecture, we refrain from adding the security architecture to the implementation. The prototype of the model is written in Java version 1.5. For further details concerning the implementation see also [Mel07].

We start by explaining how the implementation represents and stores the history. Next, we continue with an overview of the components of the implementation. Then, we present implementation-specific details of each component. Finally, we will discuss the results of the performance evaluation of the implementation.

7.1 History

The model does not specify where the history is stored and how it should be represented. As a consequence, the implementation needs to define a way how to represent and store the history. In this prototype implementation the history is split in several parts. One part of the history is stored within the corresponding document and is represented as XML data. We store the history within the documents, because this reduces the complexity of the implementation. In terms of efficiency it is better to store the history in a

separate database, because databases are better optimized for being queried. The second part of the history is the is-copy-of relation among objects. We store this part separately and described it together with the Copy Database in Section 7.2.2.

We store the history of an XML document by adding an **History** element to each existing element of the document. We avoid conflicts with the names of existing elements of the document by using an individual XML namespace (namespaces are explained in Section 2.2.3). This **History** element has two child elements. The first child element is the **Element-History** element, which stores the effects of operations that were performed on the element itself. The second child element is the **Text-History** element, which captures the effects of operations that were performed on the text content of the corresponding element.

Each part of the **History** element uses **Entry** elements to denote the effects of specific operations. An **Entry** element is composed of an **Action** element and a **Context** element. The **Action** element describes the occurred operation more precisely. For this purpose, it uses two optional arguments and a mandatory **Op** attribute, where the **Op** specifies the operation and the arguments give additional details that are specific for the operation. Table 7.1 lists the arguments of the operations defined in the model.

Operation	Op	Arg1	Arg2
Create Document	Create Element	-	-
Create Element	Create Element	-	-
Delete Element	Delete Element	-	-
Transfer Element	Create Element	-	-
Create Text	Create Text	Block ID	-
Delete Text	Delete Text	Block ID	-
Transfer Text	Create Text	Block ID	-
Create Attribute	Create Attribute	Attribute name	Attribute value
Change Attribute	Change Attribute	Attribute name	Attribute value
Delete Attribute	Delete Attribute	Attribute name	-
View (Element)	View Element	-	-
View (Text)	View Text	Block ID	-
View (Attribute)	View Attribute	Attribute name	-

Table 7.1: Arguments for storing operations in histories

We use block IDs to reference individual text blocks. In similar fashion, we also use IDs to reference subjects and roles. To store the context of an operation, we use a **Context** element with three attributes. The first attribute specifies the date when an operation was performed, whereas the

second attribute stores the role of the corresponding subject and the third attribute keeps track of the subject itself.

Figure 7.1 illustrates a **History** element. This example covers three operations. First, the element itself was created. After this, text content was added to the element. And finally, the attribute **funded-by** with the initial value **Company A** was created.

```

<History>
  <Element-History>
    <Entry>
      <Action Arg1="" Arg2="" Op="Create Element"/>
      <Context Date="2007-10-26T08:04:31.345+02:00"
        RoleID="2" SubjectID="13"/>
    </Entry>
    <Entry>
      <Action Arg1="funded-by"
        Arg2="Company A" Op="Create Attribute"/>
      <Context Date="2007-10-26T08:07:13.543+02:00"
        RoleID="1" SubjectID="67"/>
    </Entry>
  </Element-History>
  <Text-History>
    <Entry>
      <Action Arg1="0" Arg2="" Op="Create Text"/>
      <Context Date="2007-10-26T08:05:45.864+02:00"
        RoleID="0" SubjectID="15"/>
    </Entry>
  </Text-History>
</History>

```

Figure 7.1: Example of a history element

7.2 Components

All components described in Chapter 6 are implemented in the prototype. These components are depicted together with the user of the system in Figure 7.2. These components are the User Interface (UI), the Document Processor (DP), the Policy Enforcement Point (PEP), the Policy Decision Point (PDP), the User Database (User DB), the Document Database (Doc DB), the Copy Database (Copy DB) and the Rule Database (Rule DB). The arrows indicate which components communicate with each other.

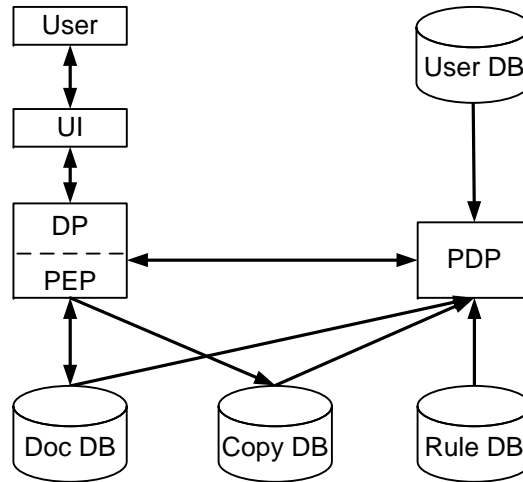


Figure 7.2: Components implemented in the prototype

Each component can be configured to communicate locally or remotely with the other components. A faster local communication method can be used, if the corresponding components are executed on the same computer. In the other case, where the components are running on different computers, a different communication method must be used. For this purpose, classes were used to abstract from the current communication method. For example, to communicate locally with the Copy Database the class `server.LocalCopyDBConnector` must be used. If the Copy DB is located on a different computer the class `server.RemoteCopyDBConnector` must be used instead. This approach allows to use the prototype on a single computer, e.g., when the system is used only by a small number of users. When the system is used by a large number of users, the components can be distributed over several computers to give more computational resources to individual components. After this overview, we discuss the components individually. We start with the User Interface.

7.2.1 User Interface

Generally speaking, the User Interface presents documents to the user and offers commands to edit these documents. The component design of the architecture allows to use any User Interface that is compatible with the interface defined in the implementation. As a consequence, the User Interface can be designed to edit specific types of data more efficiently in terms of usability. For example, the user interface can visualize the represented data in a graphical way and offer editing functions that make use of the semantics

of the edited data. The user interface of this prototype is not optimized for specific types of data and is mainly designed to illustrate the concepts of this thesis. Therefore, the current User Interface only offers a textual representation of the XML documents and the corresponding editing command are independent of the semantics of the processed XML documents. Figure 7.3 shows a screenshot of the User Interface of the prototype. We will describe the functions offered by this User Interface.

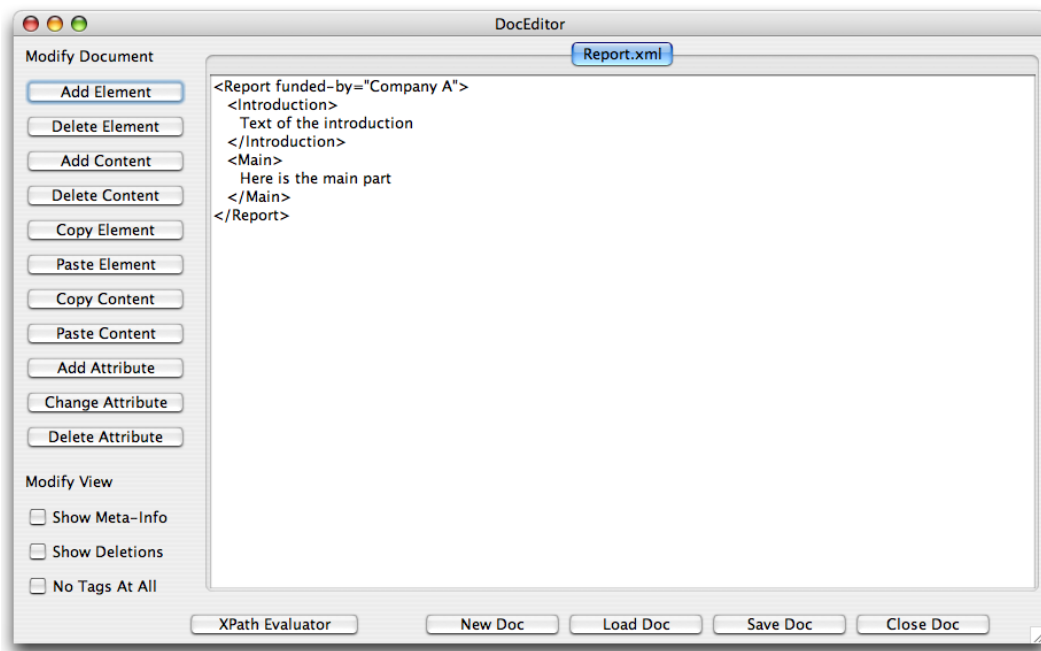


Figure 7.3: Screenshot of the User Interface

This User Interface displays an opened XML document in its textual representation in the middle area of the window. Multiple documents can be opened simultaneously, but only one document is displayed at any time. A currently not displayed document can be displayed by clicking in the top area of the window, where the name of the document is displayed. In the screenshot in Figure 7.3, only one document with the name `Report.xml` is opened.

Buttons that can be used to edit the document are on the upper left side of the User Interface window, where each such button corresponds to one operation of the model. Three check boxes that activate different view options are located below the button for editing. These check boxes can be used to illustrate the internal operations of the User Interface. The first of these buttons allows to view the histories, which are stored within the

document. The second button allows to view deleted elements. Finally, the third button hides all tags and shows the text content of the opened document.

The button **XPath Evaluator**, which is located below the document area on the left side, allows to manually enter an XPath expression that makes use of the extended functions defined in this thesis. The entered XPath expression is evaluated for the currently active document the the resulting list of nodes is displayed. This function is designed for demonstration purposes, too.

Finally, four buttons that are used to create a new document, to load an existing document, to save a changed document and to close an opened document, are located on the right side of the **XPath Evaluator**. The loading of a document performs the check-out, as it is described in Section 5.2.1 of Chapter 5. In similar fashion, the storing of a document triggers the check-in mechanism. Both mechanisms are used to efficiently reduce the number of required view recalculations.

7.2.2 Copy DB

In this implementation, we store the rules in an XML document, which is loaded when the server components of the system are started. We use **Is-Copy-Of** elements to represent individual is-copy-of relations. Within such an element, the source of a copy operation is captured by a **Source** element, whereas the destination is described by a **Destination** element. These element denote a source or destination by a sequence of document ID, element ID and block ID. In case of elements, the block ID is set to “-1”.

```
<CopyDB>
  <Is-Copy-Of>
    <Source>(0:3:-1)</Source>
    <Destination>(0:4:-1)</Destination>
  </Is-Copy-Of>
  <Is-Copy-Of>
    <Source>(1:1:0)</Source>
    <Destination>(4:3:0)</Destination>
  </Is-Copy-Of>
</CopyDB>
```

Figure 7.4: Example of the XML representation of the copy database

Figure 7.4 shows an example of such an XML representation. This example features two is-copy-of relations. The first relation reflects the transfer of

an element within a document with the document ID 0, whereas the second relation describes the transfer of a text block from the document with the ID 1 to the document with the ID 4.

7.2.3 Rule DB

The Rule DB stores the rules in an XML document, which is loaded when the server components are started. Such a policy file, must specify a default policy which is applied when no other rule is matching. Moreover, it support two types of rules, namely unary rules and transfer rules.

```
<Policy>
  <Default>
    <Mode>Deny</Mode>
  </Default>
  <Rule Type="Unary">
    <Role>1</Role>
    <Operation>Change Attribute</Operation>
    <Object>//@funded-by</Object>
    <Mode>Allow</Mode>
  </Rule>
  <Rule Type="Copy">
    <Role>3</Role>
    <Source>/Report[@funded-by="Company A"]</Source>
    <Destination>/Report[@funded-by="Company B"]</Destination>
    <Mode>Allow</Mode>
  </Rule>
</Policy>
```

Figure 7.5: Example of a policy document

Figure 7.5 shows an example of such a policy document. In this example, the default policy is “deny”. Moreover, this example shows a unary rule as well as a copy rule.

7.2.4 Policy Enforcement Point

The Policy Enforcement Point intercepts each performed operation and sends a request to the Policy Decision Point to ask whether the operation in question is allowed. For that purpose, the Policy Enforcement Point offers a method for every operation defined by the model. The Policy Enforcement Point can be configured to communicate with different PDPs. However, one

specific PDP must be chosen. This configuration is made with the **Server Configurator**, which we describe in Section 7.3.

7.2.5 Policy Decision Point

The Policy Decision Point loads the access control rules when the component is initialized. A path must be specified where the XML file containing the access control rules can be found. After initialization, the Policy Decision Point can create a view of a document for a specified user and evaluate the access control rules to check whether a specific operation is allowed.

7.3 Configuration

The server components can be configured by a dedicated **Server Configurator** program. This program allows to define which classes are used for specific functions. For example, this allows to switch between a local or remote Copy DB. Moreover, the implementation of specific components, e.g., the Copy DB, can be replaced by a different implementation that support the defined interface. This allows to easily exchange components of the current prototype without the need to modify the existing source code. In addition to this, the **Server Configurator** allows to define where the Copy DB is stored, where documents are stored and where the policy document is stored.

Figure 7.6 shows a screenshot of the **Server Configurator**. In this example, the prototype is executed locally and the paths to all required files are set to a **HiBac** directory within the home directory of the user **proeder**. The server uses a directory where all XML documents are stored, which is referred to as “document folder” in the **Server Configurator**. In addition to the directory, the server uses a text file to store the names of all XML documents together with their document IDs. In the screenshot, the corresponding file is named `docNames.txt`.

7.4 Performance Evaluation

In this section, we summarize the results of the performance evaluation, that is described in detail in [Mel07]. As the performance of XPath queries has already been discussed, e.g., in [GK02, GKP03, GKP05], we only evaluate the performance of the functions that we have added to XPath. Moreover, we measure the time to calculate a view to determine the impact of our extension functions on the overall execution time of an XPath query.

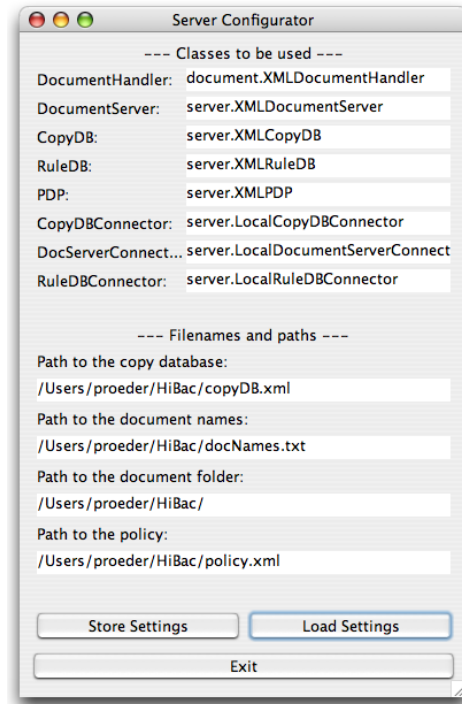


Figure 7.6: Screenshot of the Server Configurator

To determine the runtime behavior of an XPath function, we have tested several different factors to check whether the runtime depends on one or more of these factors. We learned from our tests, that the runtime of each function depends on only one specific factor, e.g., the number of nodes of the current document. Moreover, we see that groups of functions have identical runtime behavior and depend on the same factor. These groups of similar runtime behavior do not exactly match the categories that we have defined in Section 4.7. After we have identified the factor on which the runtime of a specific function depends, we made a series of measurements where we increased that factor from measurement to measurement. Each individual test was repeated five times to reduce random sources of errors. With this method, we determined how the runtime depends on the factor. For example, some tests showed that functions have a constant runtime, whereas other functions have a runtime that increases linearly with a specific factor. The specification of our test system is listed in Table 7.2.

Processor	AMD Sempron 3100+
Memory	480 MB
Operating System	Microsoft Windows XP Home Edition SP2
Java Version	JRE 1.5.0 (http://java.sun.com)
XPath 2.0 Implementation	Saxon-B 8.8 (http://saxon.sourceforge.net)
Development Environment	Eclipse SDK 3.2.2 (http://www.eclipse.org)

Table 7.2: Specification of the test system

7.4.1 Performance of Individual Functions

The first group of functions of similar runtime behavior, is the group of functions that return exactly one specific node or one atomic value, e.g., a string. These functions of this group are `selfAt`, `parentAt`, `rootAt`, `getCreationContext`, `currentNode`, `srcNode`, `destNode`, `currentSubject`, `currentRole` and `isDeleted`. These functions show a constant runtime independent of the tested documents, e.g., independent of the number of nodes in the document or the size of the tested documents. The runtime of each of this functions is around 5 milliseconds on the system that was used to perform the tests. These tests show that the performance of these functions is acceptable to be used in practice.

The next group of functions of similar runtime behavior is the group that depends on the number of copies of the tested node. These functions are `copies`, `predecessors` and `successors`. Our tests show, that the runtime of these functions depends in a linear fashion on the number of copies of the tested node. For example, with 25 copies such a function takes 100 milliseconds and with 100 copies it takes about one second to compute. These tests show, that the implementation is sufficiently fast for documents where single elements are copied less than 100 times. We believe that the performance of this group of functions is still acceptable for practical usage.

The runtime of the next group of functions is determined by the number of nodes that are in a specific relation to the node used in the test. For example, the runtime of the function `childrenAt` depends on the number of child nodes of the tested node, whereas the runtime of the functions `descendantAt` depends on the number of descendants of the tested node. In all cases, the runtime depends in a linear way on the number of nodes that are in a specific relation to the node used in the test. The functions of this group are `childrenAt`, `followingAt`, `precedingAt`, `descendantAt`, `precedingSiblingAt` and `followingSiblingAt`. The call of a function of this group that returns 25 nodes takes about 100 milliseconds, whereas a call that returns 100 nodes takes about 500 milliseconds. Whether the perfor-

mance is sufficiently fast for practical usage, depends both on the documents that are used and how rules are expressed. In some cases, the performance can be too slow, to use the current implementation in practice.

The functions of the next group of similar runtime behavior are used to retrieve the context of specific actions, e.g., the changing of attribute values. The functions of this group are `getAttrChangeContexts`, `getDeletionContexts` and `getViewContexts`. The runtime of these functions depends on the number of contexts that are retrieved by the corresponding function and increases linearly with the number of contexts. In absolute numbers, retrieving 100 contexts takes about 10 milliseconds, whereas retrieving 400 contexts takes about 30 milliseconds. For practical usage, this performance is absolutely sufficient.

Group of Functions	Runtime behavior	Usability
<code>selfAt</code> , <code>parentAt</code> , <code>rootAt</code> , <code>getCreationContext</code> , <code>currentNode</code> , <code>srcNode</code> , <code>destNode</code> , <code>currentSubject</code> , <code>currentRole</code> , <code>isDeleted</code>	constant	highly usable
<code>copies</code> , <code>predecessors</code> , <code>successors</code>	linear	acceptable
<code>childrenAt</code> , <code>followingAt</code> , <code>precedingAt</code> , <code>descendantAt</code> , <code>precedingSiblingAt</code> , <code>followingSiblingAt</code>	linear	limited
<code>getAttrChangeContexts</code> , <code>getDeletionContexts</code> , <code>getViewContexts</code>	linear	highly usable
<code>created</code> , <code>viewed</code> , <code>changedAttribute</code> , <code>deleted</code> , <code>accessed</code> , <code>modified</code>	linear	needs improvement

Table 7.3: Summary of the performance of different groups

The functions of the last group with similar runtime behavior, are used to retrieve accessed nodes. These functions are `created`, `viewed`, `changedAttribute`, `deleted`, `accessed` and `modified`. The runtime of this functions increases linearly with the number of accessed nodes. For example, a function call which retrieves 100 nodes takes about 500 milliseconds, whereas a call returning 650 nodes takes about 5 seconds. The functions of this group need further improvement to be used in a real life scenario. As the current prototype is not optimized towards performance, we see large potential to increase the performance of the implementation. Finally, in Table 7.3 we

summarize the performance of the different groups of functions of similar runtime behavior.

7.4.2 Performance of the Creation of Views

We finally present the results of the performance evaluation for creating a view. For this test, we used a set of five rules, where three of the rules used extension functions that access history information. The time to create a view increases linearly with the number of nodes of the document in question. In our test case, it took 2 seconds to create a view of a document with 4000 nodes, whereas it took about 7 seconds for a document containing 12000 nodes. We think that this performance is still acceptable for being used in practice. Nevertheless, the performance can be increased with further optimizations. In addition to this, there are much faster machines than the machine that we have used for our tests.

Chapter 8

Related Work

In this chapter, we describe work that is related to this thesis. We divide the related work in three areas, which we describe in the following. Access control can be performed on the server only, which we refer to as *Server-side Access Control*. This is also the first area of related work that we discuss in Section 8.1. Server-side access control controls access only when information is released to the client. As a consequence, there is no access control after the releasing of information.

The counterpart of server-side access control is *Client-Side Access Control*, which we discuss in Section 8.2. In client-side access control, access control is only performed on the client. This type of access control is also referred to as *Digital Rights Management* or simply *Rights Management*. The client can be manipulated easily, since it is under the control of the user, who can have unlimited access to the machine. Therefore, additional mechanisms are required to avoid or to detect these manipulations. We also discuss these mechanisms, which are similar to the mechanisms that we use in our security architecture in Chapter 6.

The combination of server-side access control and client-side access control is referred to as *Usage Control* [PS04]. Usage control protects the information at any time during its entire life cycle. To protect the information while it resides on the client, the same mechanisms are required as in client-side access control. Also usage control adds additional aspects to access control, namely provisions and obligations [HBP05]. Provisions are conditions that must be satisfied before access is granted, while obligations must be fulfilled after access was granted. We discuss both concepts as well as related work in the area of usage control in Section 8.3.

8.1 Server-side Access Control

Server-side access control is the oldest form of access control. Here, access is only controlled when the information is released. After it has been copied, there is no more access control. Our model performs usage control, since documents are protected during their entire life-cycle. But since server-side access control is also one aspect of usage control, we discuss related work in the area of server-side access control in the following.

8.1.1 “Secure and Selective Dissemination of XML Documents”

The model proposed in [BF02] supports selective access definition to portions of XML documents based on their semantic structure. Access can be defined for different nodes of the document together with propagation options, which specify whether the tree below the node in question is included. Regarding these aspects, the model is very similar to our work. However, the supported operations and their semantics are different, since our approach is able to differentiate between objects with different histories. The support of data transfers differs from our work, since the model supports only a push of different views of a document to different sets of users, whereas our model allows us to define which elements of a document may be reused in other documents. In addition to this, [BF02] uses a self-defined language to specify the objects in an access control rule. This self-defined language is less expressive than XPath, since its expressions have a fixed structure and do not allow the arbitrary composition of conditions.

Moreover, in [BF02], the smallest unit of protection is an XML element, which allows to define access in a fine-grained manner. Our model goes a step beyond this by defining smaller units of protection, namely the text blocks (see also Section 4.3.3). As a consequence, our model allows even more fine-grained access control.

Summing up, our model can express conditions on the documents history, which is not possible in [BF02]. This enables a wide range of additional policies that can be formulated as access control rules with our model.

Similar approaches to [BF02] can be found in [DCPS00, DdVPS02, MTK03, GB02], where [MTK03] and [GB02] consider access control rules for the read operation only. All these approaches consider the XML element as the smallest unit of protection, in contrast to our approach, which is capable of handling parts of the text content. None of these models is capable of defining access depending on the history of a document.

8.1.2 “X-GTRBAC: An XML-Based Policy Specification Framework and Architecture for Enterprise-Wide Access Control”

In [BGBJ05], Bhatti et al. describe their framework “X-GTRBAC”, which is designed for specifying access to XML documents. They refer to their work as a framework instead of an model, since some aspects are left unspecified and need to be specified before the framework can be applied. One such aspect is the set of operations. X-GTRBAC is based on GTRBAC [JBLG05], which is build upon on TRBAC [BBF01]. TRBAC itself is designed as an extension to RBAC [SCFY96].

TRBAC has extended RBAC with so-called temporal constraints, which are constraints that depend on the current time. TRBAC is limited to specify temporal constraints only on the role activation and deactivation, but not on the user-role or permission-role assignments, which both is needed to define roles more dynamically. Finally, X-GTRBAC enhances GTRBAC with support for XML in two aspects. X-GTRBAC defines access for XML documents and also uses XML to denote this.

In contrast to our model, which focusses on how to specify the objects of the system, X-GTRBAC has a strong focus on the subjects of the system and how to dynamically change the roles of the system. Concerning these aspects, X-GTRBAC uses a similar approach as our model. For example, concerning the user-role assignment, the set of users of a role can be defined by a condition using the properties of users. This is similar to what we do with objects in access control rules, where we specify the applicable objects by their properties. We use the current content of objects and the information stored in the history to define conditions for objects in our model. X-GTRBAC uses properties of users like their age, their experience level, their qualifications or the region where they live. For example, it can be defined that a user must have a PhD and must be older than 35 years to act in the role *Design Manager*.

Another aspect that is different from our model is that X-GTRAC uses a self-defined language for specifying conditions. For that purpose, X-GTRAC uses the language *X-Grammar*, which follows the notation from the Backus-Naur-Form (BNF). In addition to regular BNF, X-Grammar allows to specify tags with attributes. This addition makes it possible to automatically translate specifications in X-Grammar to an XML Schema. In our model, we use XPath as basic language to formulate conditions, since its clearly defined semantics makes the interpretation of the resulting rules unambiguous. Moreover, XPath has a large predefined set of built-in functions that are

required for access control. In contrast to this, in the X-GTRAC framework every required function has to be specified and implemented.

8.2 Client-side Access Control

Our security architecture of Chapter 6 and the User Interface of our architecture perform client-side access control. This is related to *Digital Rights Management* (DRM), which is an approach to prevent illegal distribution of paid content. In contrast to DRM, our security architecture focuses on the protection of confidential documents. Both have in common, that access control mechanisms are also applied on the side of the client, instead of only using access control on the server that stores the information. Therefore, we discuss approaches for digital right management in Section 8.2.1.

The client can be manipulated easily, since it is under the control of the user, who can has unlimited access to the machine. Therefore, additional mechanisms are required to avoid or to detect these manipulations. We also discuss these mechanisms, which are similar to the mechanisms that we used in our security architecture in Chapter 6. As a consequence, we discuss mechanisms to detect unauthorized modifications in Section 8.2.2.

A special aspect of this topic is *Integrity Reporting*, which describes protocols how to report the system state to an attester. In our security architecture, we use an enhanced protocol for Integrity Reporting [STRE06], which is robust against a special type of attack. Accordingly, we discuss related work in the area of integrity reporting in Section 8.2.3.

8.2.1 Digital Rights Management

CIPRESS [IGD01] builds a local quarantine zone by encrypting all local files with a machine specific key, which can optionally be stored on a tamper resistant hardware module, the Elkey crypto board [CI06]. CIPRESS offers only coarse grained access rights, since there is no access control on application level. Moreover, the security of CIPRESS is based on the assumption that the client system is not compromised, since it offers no mechanism to detect or to handle a compromised system state.

Microsoft's Rights Management Services [Cor03] offer much more fine-grained access rights compared to CIPRESS. These access rights are embedded in a signed usage license, are transferred together with the document, and are enforced by a local client software, which is assumed to be not compromised. Again, an unauthorized modification of the client system is neither

detected nor handled. Moreover, the encryption keys are stored together with the encrypted documents without further protection.

In the Display-Only File Server (DOFS) architecture [YC04], all documents remain on the server and are accessed by executing the corresponding applications on the server. Only the display content of these applications is transferred to the client using windows terminal services. DOFS offers only coarsely grained access rights, since there is no access control on the application level. The approach is limited to usage scenarios with a permanent connection to the DOFS server. Moreover, a modified client is neither detected nor handled.

Besides these three approaches, there are many others such as [Inc04] or [Aut01]. All of these lack mechanisms to detect a compromised system state, which could lead to an unauthorized information transfer. The mechanisms specified by the Trusted Computing Group (TCG) provide a possible solution for the described problem. We discuss these mechanisms in the next section.

8.2.2 Detecting a Compromised System State

In the following, we list some related research projects, which make use of the TCG mechanisms to detect and handle unauthorized system modifications. We use some of their results for our security architecture in Chapter 6.

TrustedGRUB [App06] is a bootloader that uses the TPM to extend integrity measurements to the bootloader and the OS kernel. Perseus [PRS⁺01] and Turaya [LP06] are microkernel-based trusted operating systems that can be combined with TrustedGRUB to establish a trusted computing base (for details see [Pea02]). The Bear/Enforcer project [MSWM03] includes a TPM-enabled Linux Security Module (LSM) to compare hash values of applications with reference values. A similar approach is used in the Integrity Measurement Architecture (IMA) [SZJvD04] developed by IBM, which performs integrity measurements for all started processes to enable a remote attestation. The concepts provided in [Rei04] are very similar to our security architecture, since they determine access rights of the client depending on the result of the remote attestation. In contrast to security architecture, this work does not focus on the protection of XML documents.

8.2.3 Integrity Reporting

Since the specifications of the TCG are still in progress, there are still many open issues. For example, there is a large number of work focusing on the concepts of trusted computing. One such work is [SZJvD04], which presents a comprehensive prototype based on trusted computing technologies. In

particular, it is an architecture for integrity measurement, which contains an integrity reporting protocol.

Terra [GRB03] is an approach for remote attestation. It supports the integrity measurement of virtual machines providing runtime environment for sets of processes. The approach does not build up on TPM and does not provide a protocol for integrity reporting to remote entities, which are the main differences to our work.

Our security architecture is based on the assumption that a trusted operating system measures all executed code. This concept is also referred to as binary attestation, because the binary files of the executables are measured. In contrast to that, [HCF04, SS04] focus on semantic attestation based on attesting the behavior of software components. However, the idea of attesting properties instead of binaries is very appealing, but the problem is to define and measure these properties. There is still a lot of research required until the concept of property-based attestation can be used in practice. Nevertheless, once all remaining problems are solved, the protocols of our security architecture can easily be enriched with this approach.

The authors of [BLP05] propose the integration of key exchange protocols into Direct Anonymous Attestation (DAA) [BCC04] in peer-to-peer networks, which is basically similar to the attestation protocol of our security architecture. However, the objectives of the integration of key exchange protocols are different, since [BLP05] aims at building stable identities in peer-to-peer networks. Additionally, the presented approach does not feature integrity reporting and can not be directly applied to remote attestation.

Another related work is [GPS06] which aims at building secure tunnels between endpoints. But this approach adds a new platform property certificate, which links the attestation identity key to the TLS certificate. Moreover, the presented approach focuses on server attestation, which needs in turn an additional trusted certificate authority that offers the platform property certificate. In contrast to that, our approach focuses on client attestation without an additional trusted certificate authority, since we directly bind the cryptographic channel to the attestation identity key.

8.3 Usage Control

Usage control combines server-side access control with client-side access control. As a consequence, the data is protected during its entire life-cycle. In addition to regular access control, usage control also adds the aspects of provisions and obligations [HBP05] to access control. A provision is a condition that must be satisfied before access is granted, whereas an obligation is a

condition that must be fulfilled after access has been granted. For example, a provision can be that the user must sign a contract that defines how the access data must be used, e.g., a non-disclosure agreement. However, only the signing of the agreement can be enforced technically. Whether the user behaves as he signed in the contract is up to him. Another example for a provision is the requirement to record the operation in a log file. An example for an obligation is to delete the accessed data after a certain time or to only use the data in a specified way, e.g., for a specified purpose.

Our model does neither include provisions nor obligations, because we believe this will distract from our intended focus. Furthermore, the addition would cause an increased complexity and would reduce the clarity of our model. Finally, as stated in [HBP05], obligations and provisions are an optional concept of usage control, which can be added in case it is needed. Nevertheless, both concepts can be included in our model by adding corresponding fields to our access control rules (see also Section 4.6).

In the following, we discuss related work in the area of usage control. We will present different approaches and compare them with our model.

8.3.1 “Relevancy-based Access Control”

Iwaihara et al. allow to define access based on the version relationship of documents and elements among each other [ICAW05]. They refer to their approach as “Relevancy-based Access Control”, which is similar to our approach in some aspects. They refer to the approach as being relevancy-based, since they capture how versions of a document depend on each other. As a consequence, some parts of another document version of a document can be relevant for the current version. In the following, we explain how [ICAW05] captures which elements are relevant for each other.

The version relationship of documents and elements among each other is captured by two graphs: the delta version graph and the element version graph. The delta version graph describes the relation of different document versions among each other, where each edge denotes the delta between two versions. A delta is a non-empty set of the operations defined by the model. The element version graph describes the relations among the elements of the different versions of documents. There are three possible relations, which are denoted by edges labeled with the letters ‘r’, ‘u’ and ‘n’. An edge labeled with ‘r’ states that an element was replaced by another element, whereas an edge labeled with ‘u’ denotes that the content of an element was updated, where an update can represent any kind of modification. Finally, an edge labeled with ‘n’ (no change) either denotes that no operations was applied on an element within a document or that the element is the result of a copy

operation from another document.

They define six operations including *copy*, which is similar to our *copy* operation, but can only be applied to elements or subtrees and not to the text content of an element or parts of its text content.

In contrast to our model, the modification of the content of an element is modeled by the operation *update* only, which describes that the entire content of a node is replaced with a new content. Iwaihara et al. only consider read and write operations and do not define a copy operation as part of their privileges. Consequently, they can not express which transfers among documents are permitted or denied. This is a big advantage of our model, since we can define which data transfers are allowed or denied between different documents.

Moreover, they do not have the concept of splitting copied elements to have different history information for parts from different sources. Doing so, Iwaihara et al. lose important information, which cannot be used for access control.

To define the objects of a policy tuple Iwaihara et al. extend XPath by a set of functions providing access to the element version graph. These functions enable the traversing of the edges of the element version graph and offer a similar functionality as our predecessor, successor and copies functions. This approach allows to express a subset of the policies that we can express. As we record more details, we can formulate many policies which can not be expressed in [ICAW05] language. For example, if the text of an element is composed by copying text from different sources, our model captures every detail of this process and allows to express policies based on any of that information. In [ICAW05], this process is only described with an update operation and all further details are not considered. Especially, they do not differentiate whether information was created from scratch or whether text parts were copied from somewhere else. As a consequence, they can not keep track of data flows as we do.

Additionally, they define a function to inspect the time stamp of an element. It is not clearly defined what the time stamp is, but it seems that it is set to the current time whenever an element is created. It is not clear, whether the update operation affects the time stamp. The time stamp is similar to our concept of context information. In addition to time, we also maintain the subject name and the role in which the subject performed an operation. This additional information enriches the expressiveness of the rules of our model, where we can permit or deny access depending on the subject or role of a subject that performed a previous operation.

Finally, the model in [ICAW05] does not distinguish between elements and attributes and consequently does not exploit the different semantics of

both types, as we do. In our model, we offer a set of functions to inspect the former values of attributes, which is required to express certain types of policies.

Moreover, the distinction between elements and attributes allows us to gain a higher level of efficiency, since we can save resources by only keeping track of the former values of attributes, which we consider to contain the security-relevant properties of a document.

Iwaihara et al. do not include obligations or provisions as part of their model. Moreover, they do not present a system architecture and an implementation. They also do not discuss security mechanisms for their model.

8.3.2 “Controlling Access to Documents: A Formal Access Control Model”

The model described in [SBO06] defines access control for structured data. The generic document format the present is similar to XML documents, as a document in their case is a rooted tree of elements, which can also have attributes. Similar to XML, elements have a name and a text content, whereas attributes have a name and a value. In similar fashion to our model, [SBO06] also uses the role-based approach to model subjects.

Concerning the definition of the objects, there is a big difference to our model. Instead of describing objects by their properties, as we do it in our model, [SBO06] states that the owner of each object must define the policy for each individual object manually. We regard this as a big drawback, since the manual definition of access rights for individual objects is error-prone and time-consuming. In contrast to this, our model offers a method to automatically derive the access right of an objects from its properties. This saves time and reduces the chance of possible mistakes.

The basic idea of the model in [SBO06], is the concept of *sticky policies*. This idea defines that when an object is copied to a new location, its policy is also copied to the new location. This concept of sticky policies is illustrated in Figure 8.1

The model in [SBO06] also allows to copy elements and attributes. Instead of using a single operation for that purpose, the copying is defined by a pair of a copy and a paste operation, where the copy operation is used to select the source and the paste operation defines the destination and performs the actual copying. As an intermediate buffer for the copy operation, a so-called *clipboard* is used. We also the pair of copy and paste as part of our User Interface to define the source and the destination of a copy operation, but our model uses only one operation for that purpose and we do not need

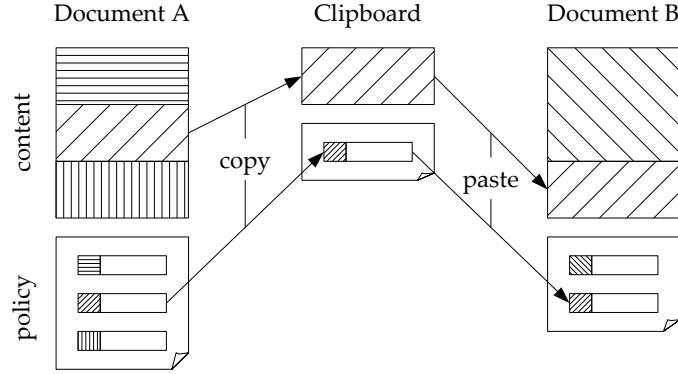


Figure 8.1: Sticky policies [SBO06]

an intermediate buffer.

Moreover, the model in [SBO06] only defines positive permissions, in contrast to our model, which also allows to specify negative rules. Having both positive and negative rules allows to specify policies much more flexible, as exceptions from a given policy can be specified more easily.

In addition to permissions, the model in [SBO06] allows to specify two types of obligations, which are log and sign. The log obligation specifies that the corresponding operation must be logged. In our model, we log every operation in the history. The sign obligation specifies that the user must sign a given contract, e.g., an agreement how to use an object, before he can access it. The signed contract is not enforced in a technical way and it is up to the user to behave as he agreed to.

Furthermore, the model in [SBO06] allows to specify environmental conditions, which must be satisfied to perform an access. These environment conditions are similar to our conditions based on the context of an operation. In [SBO06] currently only conditions depending on the current time are supported, whereas our model also allows to specify conditions on the context of a previous operation, which enables a wide range of policies that can not be specified in [SBO06]. For example, it can not be specified that in a distributed editing process a subject in a senior role has the authority to declare part of the editing as being final. In our model, such policies can be enforced.

Currently, the model specified in [SBO06] does not feature a language in which its policies can be specified. Also there is no system architecture, no security architecture and no implementation.

Summing up, the main difference to our work is that objects are specified manually by the owner of an object. In contrast to this, our model allows

to specify objects automatically by their properties, which also includes the history of an object. This allows to specify access in a more expressive and flexible way, where policies for new objects can be automatically derived from the properties of the objects.

Chapter 9

Conclusions and Future Work

In this chapter, we conclude this thesis by summarizing its main contributions. In addition to this, we present possible options for future work.

9.1 Conclusions

In this thesis, we presented a model for history-based access control, which allows to specify the objects of an access control by their properties. Our model allows to define conditions based on the object's content as well as on its history. The big advantage of this approach is that the permissions for objects can be automatically derived from their properties, which is less error-prone and less time-consuming compared to manually define access rights for individual objects.

There are many other models, e.g., [BF02, DCPS00, DdVPS02, GB02, MTK03], which also allow to specify objects by their content, but our model is much more expressive, since we also can define objects based on their history. For example, this allows to define objects by the origin from where they were copied, which is not possible in the other models. As another example, we can specify Chinese-Wall policies in a way that is better suited for real world scenarios than the original Chinese-Wall model [BN89], since our model allows to avoid the unnecessary restrictions of the original model.

To be able to record many details of the editing process, we introduced a set operations, which has a higher level of abstraction than the operation in many other models. For example, we can record that text content is composed by copying it from different sources.

In addition to this, we use a finer granularity than XML elements for access control, since we have introduced text blocks, which are parts of the text content that differ in their history. Our model automatically keeps track

of these text blocks, e.g., it splits a text block when a part of it is copied to a different location. A finer granularity is always helpful, since it allows to define the objects more precisely.

Moreover, we have both positive and negative rules, which allows in combination with our conflict resolution strategy to specify exceptions to given policies very efficiently, since every exception can be specified by an additional rule without the need to modify the existing rules.

Since we have defined our model on an abstract level, we have developed a system architecture in that supports our model in a real world scenario where multiple users concurrently edit documents and each user can work on an individual computer. To define such a system architecture, we have discussed different alternative ways to design it. Within our discussion of possible architectures, we investigated both security aspects as well as performance and efficiency concerns. Finally, we have chosen an architecture that uses the client/server-approach.

Our model introduces some challenges when it should be applied in a distributed scenario. Most of these challenges is caused by the fact that the permissions of a document can depend on the content of other documents. We presented solutions for each of these challenges, which allows to implement our model in a distributed scenario. In addition to this, we discussed the most important algorithms for implementing our model, e.g., we described the algorithms for view creation and for rule evaluation.

Next, we defined the security mechanisms for our architecture. Concerning this mechanisms, we focussed on how to protect the client from unauthorized modifications, since the client is under the control of the user, which results in a high risk of these modifications. We used the concepts defined by the Trusted Computing Group (TCG) [Pea02, Gro06] to detect unauthorized modifications of the client machine. One part of our mechanisms is the so-called remote attestation, which is used to report the state of a platform to a remote attester. For this purpose, we used an enhanced version [STRE06] of the usual remote attestation protocol [SZJvD04] that avoids a common attack, namely the spoofing of a trusted system state using an additional trusted machine.

Since our security mechanisms can decrease the system's compatibility with additional software, we used virtualization techniques to run the additional software in a different compartment on the same physical machine. A so-called hypervisor ensures that each compartment can use a different execution environment, e.g., a different operating system, and that these compartments cannot affect each other.

Finally, we have implemented our system architecture in Java. We presented the details of the implementation and explained how specific concepts

were realized. For example, we explained how we represent and store the histories. The implementation includes a user interface that allows to load, edit and save XML documents. During the editing process, the histories are maintained as defined in our model. The server components of the implementation can interpret and evaluate the access control rules defined by our model. With these components, we can illustrate the feasibility of our model.

We concluded the implementation chapter with the results of a performance evaluation. In this evaluation, we measured the time it takes to execute each individual extension function depending on its input arguments and on the processed documents. We identified groups of functions, which have a similar runtime behavior. All of these groups except of one showed an acceptable runtime behavior. The group of functions for getting accessed nodes was too slow to be used in real world scenarios. As another test, we measured the time it takes to calculate a view. In our test case, it took 2 seconds to create a view of a document with 4000 nodes, whereas it took about 7 seconds for a document containing 12000 nodes. We think that this performance is still acceptable for being used in practice. Since our implementation was not focussed on optimizing performance, we are confident that the performance can be increased significantly to meet today's requirements.

9.2 Future Work

This thesis leaves room for future work. Some of this work concerns the implementation of the prototype. First of all, the security mechanisms that we have described in Chapter 6 can be integrated into the current implementation. Moreover, the performance of the implementation can be increased. For example, we currently use components written in Java to maintain the databases of our system. We expect a big increase in terms of performance when we use a relational database system like MySQL [MyS] or PostgreSQL [Pos] for that purpose.

Furthermore, there are other methods to create a view of a document. In our implementation, we use the node filtering approach, which first retrieves the entire document and then removes inaccessible nodes. A more advanced approach is referred to as *query rewriting*. This approach modifies the query, which retrieves the document, in a way that only accessible nodes are returned. This approach is more efficient than node filtering, since it avoids to retrieve inaccessible nodes. As part of future work, we could adopt the concept of query rewriting to our implementation.

In addition to the future work on the implementation, our model can be extended in some aspects. In this thesis, we focussed on methods for speci-

fying conditions on the objects. We use roles as an abstraction to define the subjects in our access control rules. These roles model one specific property of subjects, namely the tasks a subject is allowed to perform. In addition to the tasks a subject is allowed to execute, subjects can have further properties, e.g., an age, an experience level or a qualification. These additional properties can also be used in rules to specify the subjects for which a rule is applicable more precisely. As a consequence, we can extend our model, to support additional conditions on subjects.

Moreover, we do neither support provisions nor obligations [HBP05] in our model. These concepts define conditions, which must be fulfilled before access is granted or after access has been granted respectively, e.g, a condition stating that the user must sign an agreement before he can access certain data. We can extend our model by adding another field to our access control rules which specifies provisions and obligations.

In this thesis, we also support context information to be used as part of a condition in our access control rules. We can use the subject of an operation, its current role and the time of the corresponding operation to define such a condition. As part of future work, we could also use additional information in our context description. For example, if sensors for the location are available, we could include the location in our context description.

Moreover, we could use a formal notation language to define the data structures, operations and functions of our model. This could help us to give a more precise specification of our model and help to study further properties of our model. With this method, we could analyze whether all extension functions are free of side effects, whether they introduce an unwanted information flow and other similar properties.

Index

- Access Control, 11, 113
 - client-side, 113
 - server-side, 113
- Access Control Rule, 50, 53
 - copy rule, 50
 - unary, 50
- Access Matrix model, 14
 - dynamic, 14
 - static, 14
- Attack, 85
- Authenticity, 86
- Availability, 85
- Axis, 26
- Buffer Overflow Attack, 90
- Check-in, 65, 106
- Check-out, 65, 106
- Chinese Wall model, 18
- CIPRESS, 116
- Class
 - conflict-of-interest, 18
- Client, 87
- Client-Server approach, 74
- Client-Side access control, 113
- Compartmentalization, 126
- Confidentiality, 85
- Conflict resolution strategy, 12, 39, 52, 66
- Conflict-of-interest class, 18
- Constraints, 16
 - cardinality, 17
- Context, 38, 56
 - node, 26
 - object, 18
 - role, 18
- Copy DB, 64
- Copy graph, 49, 53
 - complete, 49
- DAC, 12
- Default semantics, 52
- Deny of Service attack, 91
- Dependent document, 54, 63
- Digital Rights Management, 113, 116
- Direct Anonymous Attestation, 118
- Direct Memory Access, 89
- Discretionary Access Control, 12
- DMA, 89
- Doc DB, 64
- Document proprocessor, 64
- Document Type Defintion, 24
- DTD, 24
- Element, 33
- End tag, 22
- Environment roles, 18
- Extensible Markup Language, 21
- Granularity, 12
- GTRBAC, 115
- History, 48
- Hypervisor, 126
- Inside Attacker, 86
- Integrity, 85
- Integrity Reporting Protocol, 28

- is-copy-of relation, 49, 53, 56, 64, 69, 73, 77, 102
 - Definition, 49
- MAC, 13
- Mandatory Access Control, 13
- Memory Dump, 88
- Microsoft's Rights Management Services, 116
- Mode, 12
- Namespaces, 23
- Non-repudiation, 86
- Object, 11, 42
 - Attribute, 43
 - Text block, 43
 - XML element, 42
- Object context, 18
- Object roles, 18
- Obligation, 113
- Operating System, 88
- Operation, 36, 44
 - change-attribute, 47
 - context of, 38
 - copy, 37, 46
 - create, 37, 44
 - delete, 37, 45
 - update, 37
 - view, 37, 48
- Outside Attacker, 86
- Peer-to-Peer approach, 75
- Policy Decision Point, 64, 108
- Policy Enforcement Point, 64, 107
- Principle of least privilege, 16
- Protection level, 31
- Provision, 113
- Query Rewriting, 127
- RBAC, 15, 115
- Remote Attestation, 28
- Rights Management, 113
- Role, 15, 32
 - context, 18
 - hierarchy, 16, 32
- Rule DB, 64
- Schema, 24
- Separation of duty, 17
- Server-Side Access Control, 113
- Session, 15
- Start tag, 22
- Sticky policy, 121
- Subject, 11, 38
- Swap file, 88
- System architecture, 63, 64
 - distributed, 73
- Tag
 - end, 22
 - start, 22
- TCG, 26, 93
- TCSEC, 12
- Text block, 43
- Text content, 22, 33
- Text fragment, 36
- TRBAC, 115
- Trusted Computer System Evaluation Criteria, 12
- Trusted Computing Group, 26, 93, 126
- Trusted Platform Module (TPM), 27
- TrustedGRUB, 117
- URI, 23
- URL, 23
- Usage Control, 113, 118
- User context, 18
- User DB, 64
- User interface, 64, 121
- View, 48, 66
 - of a document, 48, 66
 - Operation, 48

- Recalculation, 63
- Virtualization, 126
- Well-formed, 23
- Workflow, 64
- X-Grammar, 115
- X-GTRBAC, 115
- XML, 3, 21
 - attribute, 43
 - element, 33, 36, 42
 - Schema, 24
- XML document, 22, 31
 - view, 48
- XPath, 25, 41, 53, 55
 - pattern, 51, 53
- XPointer, 25
- XQuery, 25
- XSLT, 25

Bibliography

- [AC04] F. T. Alotaiby and J. X. Chen. A Model for Team-based Access Control (TMAC 2004). In *In Proceedings of the International Conference on Information Technology: Coding and Computing, 2004. ITCC 2004.*, volume 1, pages 450–454 Vol.1, 2004.
- [App06] Applied Data Security Group, University of Bochum. TrustedGRUB. http://www.prosecco.rub.de/trusted_grub_details.html, May 2006.
- [ATS04] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.
- [Aut01] Authentica Inc. PageRecall: The Key to Document Protection. <http://www.authentica.com/>, 2001.
- [Bas06] European Multilaterally Secure Computing Base. Towards trustworthy systems with open standards and trusted computing. <http://www.emscb.de/>, 2006.
- [BBF01] E. Bertino, P. Andrea Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, 2001.
- [BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM Press.
- [BCDP05] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca. GEO-RBAC: a spatially aware RBAC. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models*

and technologies, pages 29–37, New York, NY, USA, 2005. ACM Press.

- [BEE07] Thomas Buntrock, Hans-Christian Esperer, and Claudia Eckert. Situation-based Policy Enforcement. In *Proceedings of the 4th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'07)*, September 3–7, 2007. (to appear).
- [BF02] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Transactions on Information and System Security*, 5(3):290–331, 2002.
- [BGBJ05] Rafae Bhatti, Arif Ghafoor, Elisa Bertino, and James B. D. Joshi. X-GTRBAC: An XML-Based Policy Specification Framework and Architecture for Enterprise-Wide Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 8(2):187–227, 2005.
- [BL73] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corp, Bedford, MA, 1973.
- [BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. RFC 3986, January 2005. <http://www.ietf.org/rfc/rfc2396.txt>.
- [BLP05] Shane Balfe, Amit D. Lakhani, and Kenneth G. Paterson. Trusted computing: Providing security for peer-to-peer networks. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, 2005.
- [BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 72–84, 1992.
- [BN89] F. D. Brewer and J. M. Nash. The Chinese Wall Security Policy. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1989.
- [BPSM⁺06] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language

- (xml) 1.0 (fourth edition). World Wide Web Consortium, Recommendation REC-xml-20060816, August 2006.
- [CD99] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C recommendation, W3C, 1999. <http://www.w3.org/TR/1999/REC-xpath-19991116>.
- [CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems (3rd Edition): Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [CI06] CE-Infosys. Elkey Crypto Board. http://www.ce-infosys.com/CeiProducts_Elkey.asp, 2006.
- [CMA00] M. Covington, M. Moyer, and M. Ahamad. Generalized Role-Based Access Control for Securing Future Applications. In 23rd National Information Systems Security Conference, Baltimore, MD, October 2000.
- [Con99] World Wide Web Consortium. Namespaces in XML. <http://www.w3.org/TR/1999/REC-xml-names-19990114>, 1999.
- [Con04] World Wide Web Consortium. XML Schema Specification. <http://www.w3.org/XML/Schema>, 2004.
- [Cor03] Microsoft Corporation. Technical Overview of Windows Rights Management Services for Windows Server 2003. White paper. Technical report, Microsoft Corporation, November 2003.
- [CWP⁺00] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.
- [DCPS00] E. Damiani, S. De Capitani, S. Paraboschi, and P. Samarati. Securing XML Documents. In *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, volume 1777 of *LNCS*, pages 121–135. Springer-Verlag, 2000.

- [DdVPS02] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A Fine-Grained Access Control System for XML Documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):169–202, 2002.
- [Den76] P. J. Denning. Fault tolerant operating systems. *ACM Comput. Surv.*, 8(4):359–389, 1976.
- [DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 610–623, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.
- [FHN⁺04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, October 2004.
- [FM04] I. Fundulaki and M. Marx. Specifying Access Control Policies for XML Documents with XPath. In *SACMAT '04: Proceedings of the ninth ACM Symposium on Access Control Models and Technologies*. ACM Press, 2004.
- [GB02] A. Gabillon and E. Bruno. Regulating Access to XML Documents. In *Working Conference on Database and Application Security*, pages 299–314. Kluwer Academic Publishers, 2002.
- [GD72] G. S. Graham and P. J. Denning. Protection - Principles and Practice. In *Spring Joint Computer Reference*, volume 40, pages 417–429, 1972.
- [GK02] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 189–202, 2002.
- [GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The Complexity of XPath Query Evaluation. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 179–190, New York, NY, USA, 2003. ACM Press.

- [GKP05] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems (TODS)*, 30(2):444–491, June 2005.
- [GLLR05] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2005 CSI/FBI Computer Crime and Security Survey. Technical report, CSI, 2005.
- [GLLR06] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2006 CSI/FBI Computer Crime and Security Survey. Technical report, CSI, 2006.
- [GPS06] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *First ACM Workshop on Scalable Trusted Computing*, Fairfax, Virginia, November 2006. ACM Press.
- [GRB03] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2003.
- [Gro06] Trusted Computing Group. Trusted Platform Module (TPM) specifications. Technical report, <https://www.trustedcomputinggroup.org/specs/TPM>, 2006.
- [Han07] Holger Hanke. Kontextabhängige Zugriffskontrolle für Unternehmensdokumente. Master’s thesis, TU Darmstadt, January 2007.
- [HBP05] M. Hilty, David Basin, and Alexander Pretschner. On obligations. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, volume 3679 of *Lecture Notes in Computer Science*, pages 98 – 117. Springer Verlag, 2005.
- [HCF04] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, 2004 2004.

- [HRU76] M.A. Harrison, W.L. Ruzzo, and J. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, 1976.
- [ICAW05] Mizuho Iwaihara, Somchai Chatvichienchai, Chutiporn Anutariya, and Vilas Wuwongse. Relevancy Based Access Control of Versioned XML Documents. In *SACMAT '05: Proceedings of the tenth ACM Symposium on Access Control Models and Technologies*, pages 85–94. ACM Press, 2005.
- [IGD01] Fraunhofer IGD. CIPRESS White Paper - An Overview of the System, Its Concepts, and Implementation in the Microsoft Windows NT Operating System Family Environment. Technical report, Fraunhofer IGD, February 2001.
- [Inc04] Adobe Systems Incorporated. A primer on electronic document security - How document control and digital signatures protect electronic documents. http://www.adobe.com/security/pdfs/acrobat_security_wp.pdf, November 2004.
- [ISO86] ISO (International Organization for Standardization). Information processing — text and office systems — standard generalized markup language (SGML). ISO Standard ISO 8879:1986(E), International Organization for Standardization, Geneva, Switzerland, October 1986.
- [JBLG05] James B. D. Joshi, E. Bertino, U Latif, and A. Ghaffor. A Generalized Temporal Role Base Access Control Model (GTR-BAC). *IEEE Transaction on Knowledge and Data Engineering*, 17, 2005.
- [JPPMKA02] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 477–484, 2002.
- [KK03] Aleksandar Kuzmanovic and Edward W. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86, New York, NY, USA, 2003. ACM Press.

- [KKC02] A. Kumar, N. Karnik, and G. Chafle. Context Sensitivity in Role-Based Access Control. *SIGOPS Oper. Syst. Rev.*, 36(3):53–66, 2002.
- [KZB⁺91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.
- [Lam69] B. W. Lampson. Dynamic Protection Structures. In *Proceedings of the Fall Joint Computer Conference*, pages 27–38, 1969.
- [Lam71] B.W. Lampson. Protection. In *Proceedings of the Fifth Annual Princeton Conference on Information Science and Systems*, pages 437–443, 1971.
- [LE01] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, August 2001.
- [LP06] M. Linnemann and N. Pohlmann. Die vertrauenswürdige Sicherheitsplattform Turaya. In *DACH Security 2006*. Patrick Horster, syssec Verlag, 2006.
- [Mea99] C. Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. *csfw*, 00:4, 1999.
- [Mel07] Fredrik Mellgren. History-Based Access Control for XML Documents. Master’s thesis, Technische Universität Darmstadt, June 2007.
- [MSWM03] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, December 2003.
- [MTK03] M. Murata, A. Tozawa, and M. Kudo. XML Access Control using Static Analysis. In *ACM Conference on Computer and Communications Security*. ACM Press, 2003.
- [MyS] MySQL. <https://www.mysql.com/>.

- [Nee94] Roger M. Needham. Denial of Service: An Example. *Communications of the ACM*, 37(11):42–46, 1994.
- [Net02] Network Working Group . Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations, RFC 3305. <http://tools.ietf.org/rfc/rfc3305.txt>, August 2002.
- [Pea02] Siani Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.
- [Pos] PostgreSQL. <http://www.postgresql.org/>.
- [PRS⁺01] Birgit Pfitzmann, James Riordan, Christian Stübke, Michael Waidner, and Arnd Weber. The PERSEUS System Architecture. In *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*. Vieweg Verlag, 2001.
- [PS96] B. W. Parkinson and J. J. Spilker, Jr. *The Global Positioning System: Theory and Applications*. American Institute of Aeronautics and Astronautics, 1996. Progress in astronautics and aeronautics; v. 163-164., 1996.
- [PS04] Jaehong Park and R. S. Sandhu. The UCON_{ABC} Usage Control Model. *ACM Transactions on Information and System Security*, 7:128 – 174, 2004.
- [PSTT96] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Bayou: Replicated database services for world-wide applications. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 275–280, New York, NY, USA, 1996. ACM Press.
- [RC05] Jason F. Reid and William J. Caelli. DRM, trusted computing and operating system architecture. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, pages 127–136, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
- [Rei04] Reiner Sailer and Trent Jaeger and Xiaolan Zhang and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM confer-*

- ence on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.
- [RSGE06] P. Röder, F. Stumpf, R. Grewe, and C. Eckert. Hades - Hardware Assisted Document Security. In *Proceedings of the Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.
- [SBHE07] Frederic Stumpf, Michael Benz, Martin Hermanowski, and Claudia Eckert. An Approach to a Trustworthy System Architecture using Virtualization. In *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC-2007)*, LNCS, Hong Kong, China, 2007. Springer-Verlag. to appear.
- [SBO06] Paul E. Sevinc, David Basin, and Ernst-Rüdiger Olderog. Controlling Access to Documents: A Formal Access Control Model. In *Proceedings of the International Conference on Emerging Trends in Information and Communication Security (ETRICS 2006)*, Lecture Notes in Computer Science (LNCS), pages 352 – 367. Springer-Verlag, June 2006.
- [SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
- [SJZvD04] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.
- [SKK⁺97] C.L. Schuba, I.V. Krsul, M.G. Kuhn, E.H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a Denial of Service Attack on TCP. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 208–223, 1997.
- [SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.
- [SS94] R. S. Sandhu and P. Samarati. Access Control: Principle and Practice. *Communications Magazine*, 32:40–48, 1994.

- [SS04] Ahmad-Reza Sadeghi and Christian Stueble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, New York, NY, USA, 2004. ACM Press.
- [STRE06] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *Proceedings of the Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, November 2006.
- [SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium*. IBM T. J. Watson Research Center, August 2004.
- [Tal03] A. S. Talwadker. Survey of Performance Issues in Parallel Database Systems. *J. Comput. Small Coll.*, 18(6):5–9, 2003.
- [The91] The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding. Version 1.0. Volumes 1 and 2*. Addison-Wesley Developers Press, 1991.
- [Tho97] Roshan K. Thomas. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 13–19, New York, NY, USA, 1997. ACM Press.
- [Tic85] W. F. Tichy. RCS - A System for Version Control. *Softw. - Practice and Experience*, 15(7):637–654, 1985.
- [Uni85] United States Department of Defense. Trusted Computer System Evaluation Criteria. DoD Standard 5200.28-STD, December 1985.
- [Wal03] Dan S. Wallach. *A Survey of Peer-to-Peer Security Issues*. Springer Berlin / Heidelberg, 2003.
- [Wor06] World Wide Web Consortium (W3C). <http://www.w3.org/>, 2006.
- [YC04] Yang Yu and Tzi-cker Chiueh. Display-Only File Server: A Solution against Information Theft Due to Insider Attack. In

DRM '04: Proceedings of the 4th ACM workshop on Digital rights management, pages 31–39, New York, NY, USA, 2004. ACM Press.